View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by NASA Technical Reports Server

Grant Number NAG8-093

Efficient Parallel Architecture for Highly Coupled Real-Time Linear System Applications

by

Chester C. Carroll
Cudworth Professor of Computer Architecture

Abdollah Homaifar
Temporary Visiting Assistant Professor
of Electrical Engineering

and

Soumavo Barua
Graduate Research Assistant

Prepared for

The National Aeronautics and Space Administration

Bureau of Engineering Research
The University of Alabama
January 1988

BER Report No. 419-17

ACKNOWLEDGEMENT

This research was supported by NASA, George C. Marshall Space Flight Center, Huntsville, Alabama, under Grant Number NAG8-093 and conducted in the Computer Architecture Research Laboratory in the College of Engineering at The University of Alabama.

LIST OF ABBREVIATIONS

ABPC	Adams Bashforth Predictor Corrector
GaAs	Gallium Arsenide
PE	Processing Element
PIA	Parallel Integration Algorithm
RISC	Reduced Instruction Set Computer
TUS	Time Units
WSI	Wafer Scale Integration
FAST	Flexible Architecture Simulation Task
REMPS	Reconfigurable Multiprocessor for Scientific Supercomputing
X^P_n	Predicted value of variable X at the n^{th} computing interval
X^C_n	Corrected value of variable X at the n^{th} computing interval
x	state variable
R	Control Weighting Matrix
Q	State Weighting Matrix
H	Terminal State Weighting Matrix

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

A systematic procedure has been developed for exploiting the parallel constructs of computation in a highly coupled, linear system application. An overall top down design approach is adopted.

Differential equations governing the application under consideration are partitioned into subtasks on the basis of a data flow analysis. The interconnected task units constitute a task graph which has to be computed in every update interval. Multiprocessing concepts utilizing parallel integration algorithms are then applied for efficient task graph execution. A simple scheduling routine has been developed to handle task allocation while in the multiprocessor mode.

Results of simulation and scheduling are compared on the basis of standard performance indices. Processor timing diagrams have been developed on the basis of program output accruing to an optimal set of processors.

Basic architectural attributes for implementing the system is discussed together with suggestions for processing element design. Emphasis has been placed on flexible architectures that are capable of accommodating widely varying application specifics.

CHAPTER 1

INTRODUCTION

1.1 Background:

Real-time application algorithms are characterized by complex and time consuming computations suitable for processing in large mainframes and associated machines. However cost and space constraints would favor the development of small multiprocessor machines that are capable of exploiting the inherent parallel constructs of computation [1]. With decreasing hardware costs a large number of processors may be grouped together to form specialized processing clusters or modules [2]. Flexible customization methodology may serve to utilize these specialized hardware modules to achieve computational speeds that are beyond the limits of uniprocessor sequential methods. The vast increase in computing power accompanied by the drastic reduction in cost, makes parallel processing in multiprocessor environment a viable option for the critical timing constraints of real-time applications.

1.2 Objective:

The objective of this research is to develop a systematic procedure for evolving a computational model that is

particularly amenable for parallel processing in a multiprocessor environment. An overall top-down approach (see Figure 1.1) is adopted. Any real-time system may be represented in general by a set of differential equations which govern the dynamic behavior of the system. As a specific example, a prototype real-time control problem is modeled as a set of differential equations. These are mapped onto a task graph which is then allocated to a set of processors in accordance with an allocation algorithm. This is followed by a verification and comparison stage wherein the results of such a mapping are compared with that of traditional uniprocessor methods in terms of speed up ratio, efficiency and average processor utilization. Finally, hardware schemata are included for processors and their design.

1.3 Research Phases:

Research was conducted in the following phases:

1) Problem Identification
2) Task Graph Development
3) Scheduling and Simulation
4) Hardware and software issues

A few simplistic assumptions were made throughout the overall research. Interprocessor communication time was neglected in all cases. Although the author acknowledges that this is not a very practical assumption, the overall performance improvement would not be greatly undermined even if such delays are taken into account. Finally , an inexhaustible supply of

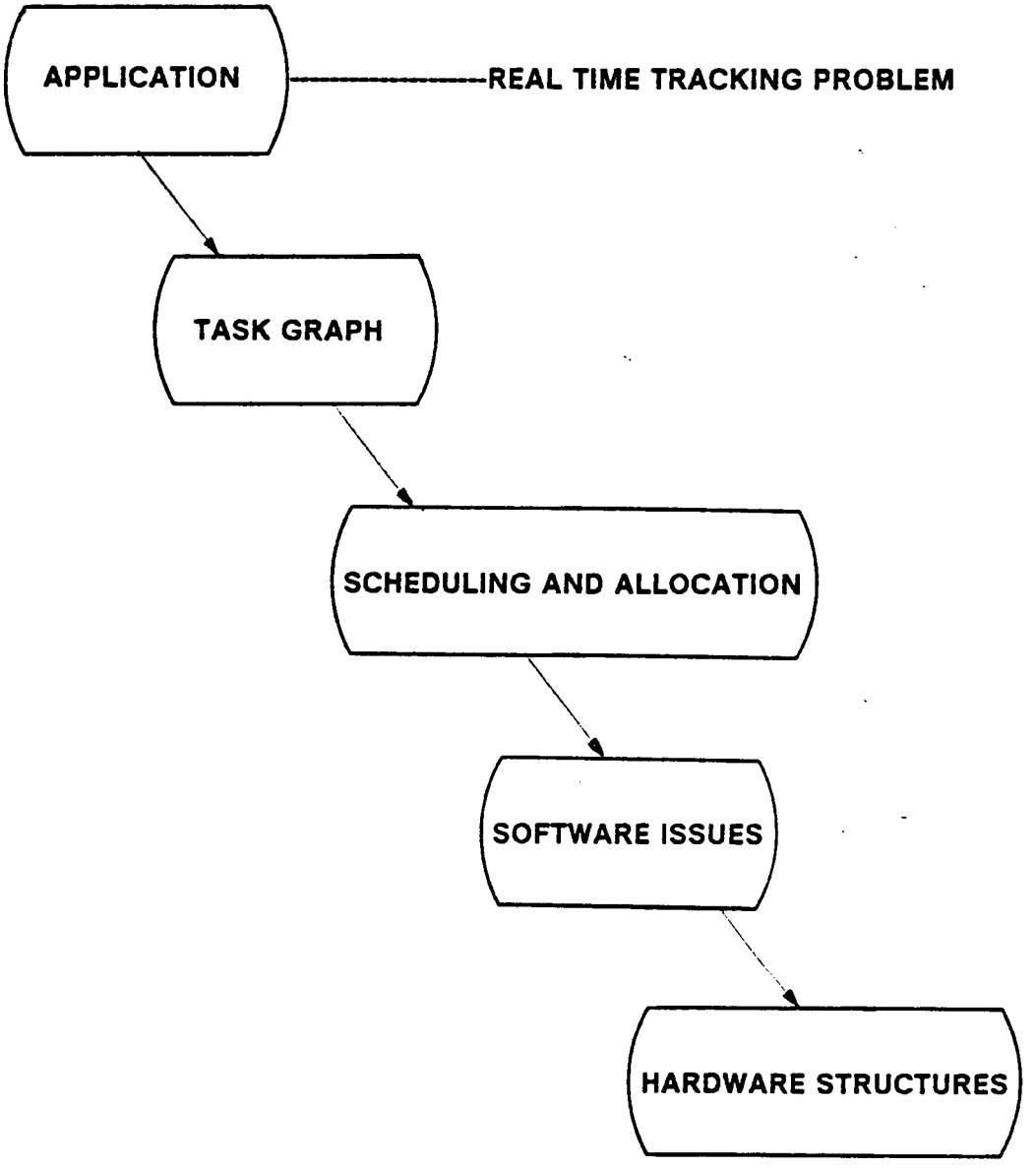


Figure 1.1 Overview of Research Project

hardware resources has been assumed. The number of available processors has been treated as a variable parameter which may be optimized to obtain maximum speed of execution. It is this singular fact that makes a flexible architecture the best hardware support for this project.

CHAPTER 2

APPLICATION AND MODEL DEVELOPMENT

A vast majority of real time control problems can be represented by a stochastic system of equations and an associated cost function or performance index. The dynamic behavior of the system is modeled by a set of linear state equations of the form:

$$\dot{x}(t)=A(t)x(t)+B(t)u(t)$$

The major objective in such a system model is to obtain the optimal control law by minimizing the overall cost function [3].

2.1 Problem Identification

A typical class of optimal control problems are of the tracking type. These are primarily concerned with constraining the motion of a body in a defined trajectory and are widely used in attitude control of rocket, missile guidance, aircraft landing analysis etc. The cost function to be minimized for optimal control is commonly represented as:

$$J=0.5[x(t_f)-r(t_f)]^T H[x(t_f)-r(t_f)]+0.5\int_{t_o}^{t_f}\{[x(t)-r(t)]^T Q(t)[x(t)-r(t)]+u^T(t)R(t)u(t)\}dt$$

Modern control theory suggests two principle ways of solving such problems (Appendix A). One convenient technique is the generation of a set of first order differential equations known as the Matrix Ricatti Differential Equations (see Figure 2.1) having a form :

$$\dot{K}=-K(t)A(t)-A^{T}(t)K(t)-Q(t)+K(t)B(t)R^{-1}(t)B^{T}(t)K(t)$$

$$\dot{s}(t)=-[A^{T}(t)-K(t)B(t)R^{-1}(t)B^{T}(t)]s(t)+Q(t)r(t)$$

It may be easily proved that if K is a "n by n" symmetric matrix and s is a "n by 1" vector , then the above matrix equations reduce to a set of "n(n+1)/2+n" first order differential equations which have to be solved in real time.With large values of "n" as is true for most practical systems , an inconveniently large set of equations is obtained. Even with available current technology, it requires a mini supercomputer to perform the necessary computations.

2.2 Solution Methods

Several standard software routines using Runge Kutta Method, Adams Bashforth Method is available for solving differential equations and may be applied to the solution of Matrix Ricatti Equation. However, these are sequential techniques with a set limitation on execution speed. By employing parallel integration algorithms (PIA) it is possible to obtain a greater throughput while maintaining the same level of accuracy [4]. The method presented here is a modified version of that proposed by Willard L. Miranker and Werner Liniger [5].

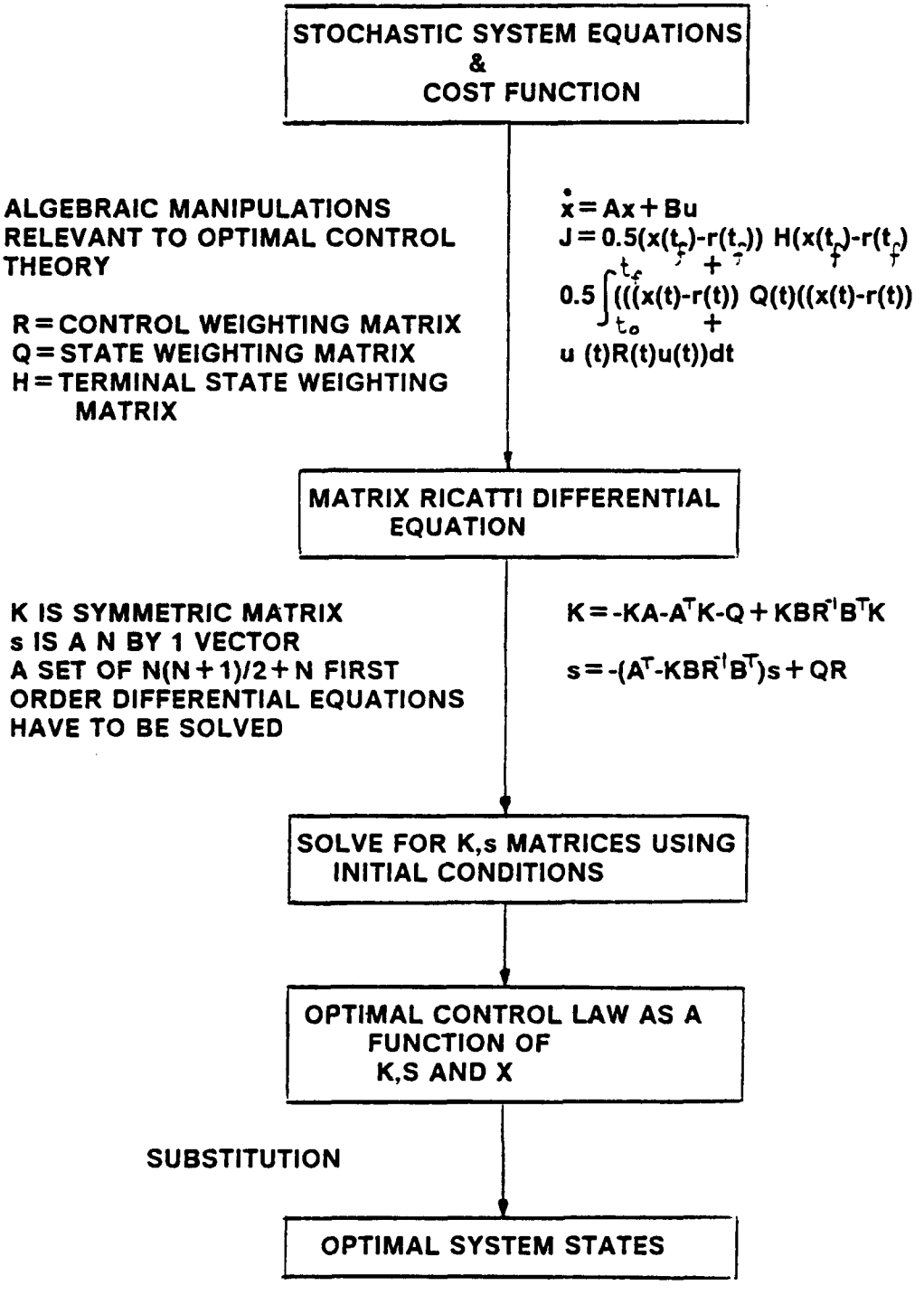


FIGURE 2.1 Overall Problem Representation

A modification is necessary as the aforementioned authors developed their algorithm for standard differential equations which are typically initial value problems as opposed to the Matrix Ricatti Equations where the integration has to be carried out backwards in time.

2.3 Parallel Integration Algorithm

A widely used technique for solving differential equations is the Adam Bashforth Predictor Corrector (ABPC) method. For a general problem of the type

$$y'=f(x,y),\ x > 0,\ y(0)= y_0,$$

the differential equations for a two step ABPC method are given

$$y^P_n = y^c_{n-1} + h/2\ [\ 3\ f^c_{n-1} - f^c_{n-2}\ ]$$

$$y^c_n = y^c_{n-1} + h/2\ [\ f^P_n + f^c_{n-1}\ ]$$

where h = step increment = x_n / (n-1);

It is apparent that the predicted value at the $(n)^{th}$ step is used in the next step to compute the corrected value at the $(n)^{th}$ step. The sequence of computation is schematized (see Figure 2.2). The "P" and "C" lines denote the predicted and corrected values of the function. A hypothetical computation front is indicated by means of a dotted line. The directed line segments display that at the $(n)^{th}$ mesh point , results flow in from both sides of the computation front thereby precluding any chances of simultaneous prediction and correction.

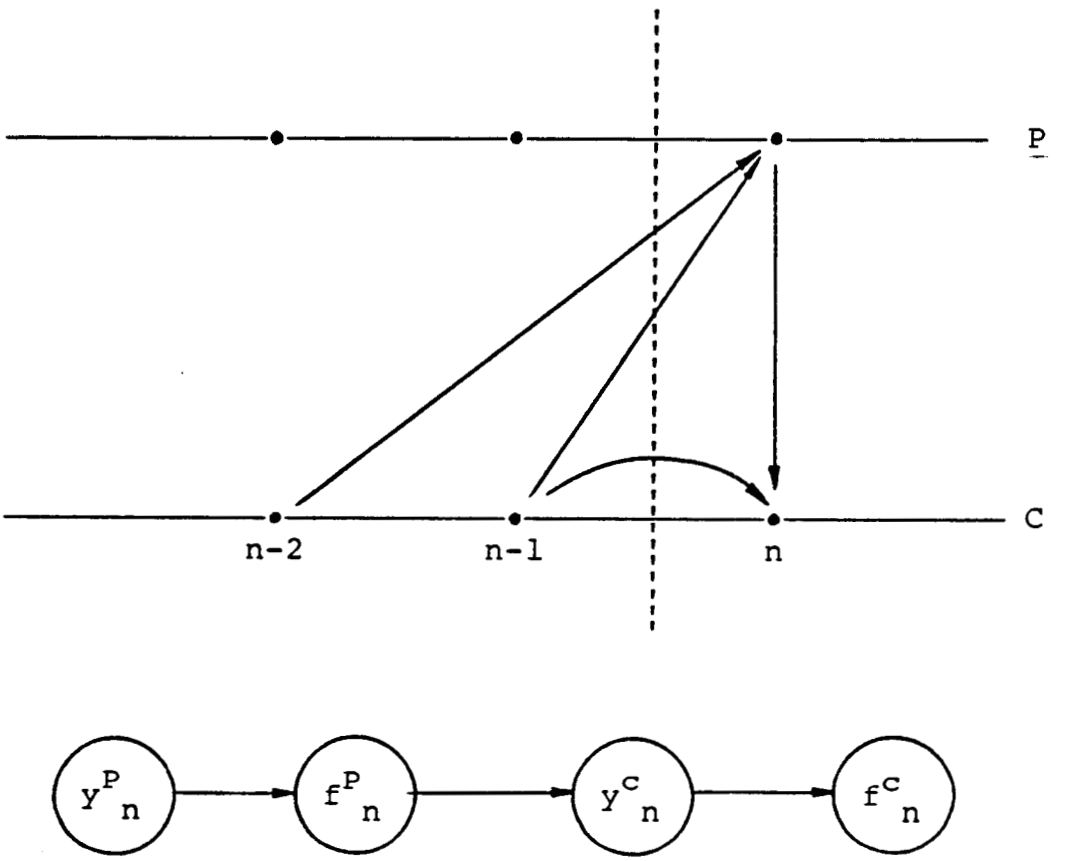


Figure 2.2 Serial Computation Sequence

A suitable modification converts this sequential technique into an effective PIA. The modified equations are:

$$y^P_n = y^C_{n-2} + 2\ h\ f^P_{n-1}$$

$$y^C_{n-1} = y^C_{n-1} + h/2\ [\ f^P_{n-1} + f^C_{n-2}\ ]$$

The computation front and associated sequence of computation are shown (see Figure 2.3). The arrows indicate that calculation at any step depends only on information at previous mesh points. This implies that the parallel implementation simultaneously accommodates prediction at the $(n)^{th}$ step and correction at the $(n-1)^{th}$ mesh point and thus may be executed in parallel on two arithmetic processors.

Application of this technique to the solution of Matrix Ricatti equations necessitates the computation front to proceed backward in time. For this purpose the aforementioned parallel differential equations are modified to yield :

$$y^P_{n-2} = y^C_n - 2\ h\ f^P_{n-1}$$

$$y^C_{n-1} = y^C_n - h/2\ [\ f^P_{n-1} + f^C_n\ ]$$

The corresponding computation front has also been shown (see Figure 2.4).

2.4 The Prototype Problem

A prototype reflects an actual problem area with all its attributes but in smaller dimensions. It provides the researcher with a congenial environment to experiment novel schemes. In

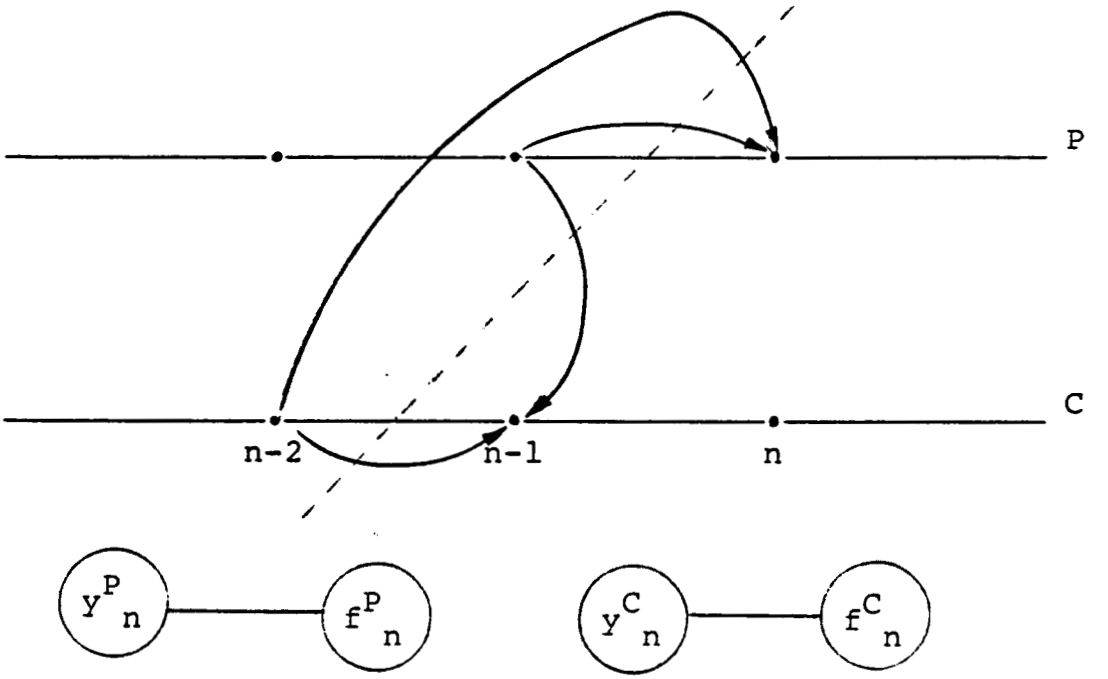


Figure 2.3 Parallel Computation Sequence

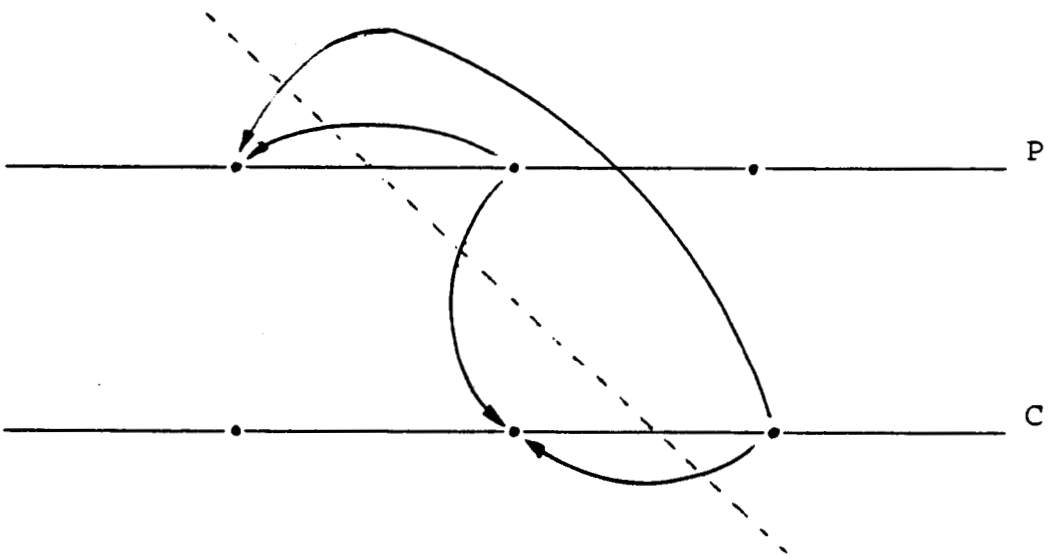


Figure 2.4 Reverse Parallel Computation Sequence

this thesis, a prototype tracking problem has been considered so as to illustrate the basic concepts and ideas that were developed in course of research.

The system to be controlled is assumed to be represented by two state equations:

$$\dot{x}_1(t) = x_2(t)$$

$$\dot{x}_2(t) = 2x_1(t) - x_2(t) + u(t)$$

The performance index to be minimized is

$$J(u) = \int_0^T \{[x_1(t) - 0.2t]^2 + 0.025u^2(t)\}dt$$

In this problem the major objective is to maintain the state x_1 close to the ramp function $r_1(t) = 0.2t$. The Matrix Ricatti equations for such a system are :

$$\dot{k}_{11}(t) = 20\ k_{12}^2(t) - 4\ k_{12}(t) - 2$$

$$\dot{k}_{12}(t) = 20\ k_{12}(t)k_{22}(t) - k_{11}(t) + k_{12}(t) - 2\ k_{22}(t)$$

$$\dot{k}_{22}(t) = 20\ k_{22}^2(t) - 2\ k_{12}(t) + 2\ k_{22}(t)$$

$$\dot{s}_1(t) = 2\ [\ 10\ k_{12}(t) - 1\ ]\ s_2(t) + 0.4t$$

$$\dot{s}_2(t) = -s_1(t) + [\ 20\ k_{22}(t) + 1\ ]\ s_2(t)$$

All the equations in the above set are cross coupled. However, the computational parallelism inherent in the equations may be exploited to obtain a higher throughput. This is discussed in the next chapter.

CHAPTER 3

PARALLEL IMPLEMENTATION

One of the important potentials of multiprocessor systems is the ability to speed up computation by concurrently processing independent portions of a given assignment [1, 11]. Extensive research is being carried out to develop mathematical models that can be solved efficiently on parallel processors [6]. The first step in developing such multiprocessor models is to identify the parallelism within the mathematical formulation of the problem. This necessitates a data flow analysis of the problem with a subsequent evolution of a " task graph ". This is then allocated to a set of processors by means of a scheduling algorithm so as to obtain minimum achievable execution time.

3.1 Task Graph Attributes

A task graph represents a set of "jobs" or "computation units" arranged in accordance with certain precedence constraints. Such a set is generally described by a "finite directed acyclic graph" [7] and is assumed to have single entry and terminal nodes through which all other nodes may be accessed. Task execution times are represented by node weights [8]. An example of a task graph is shown (see Figure 3.1).

In most practical problems, the mathematical nature of the model yields a set of closely coupled equations as is also true

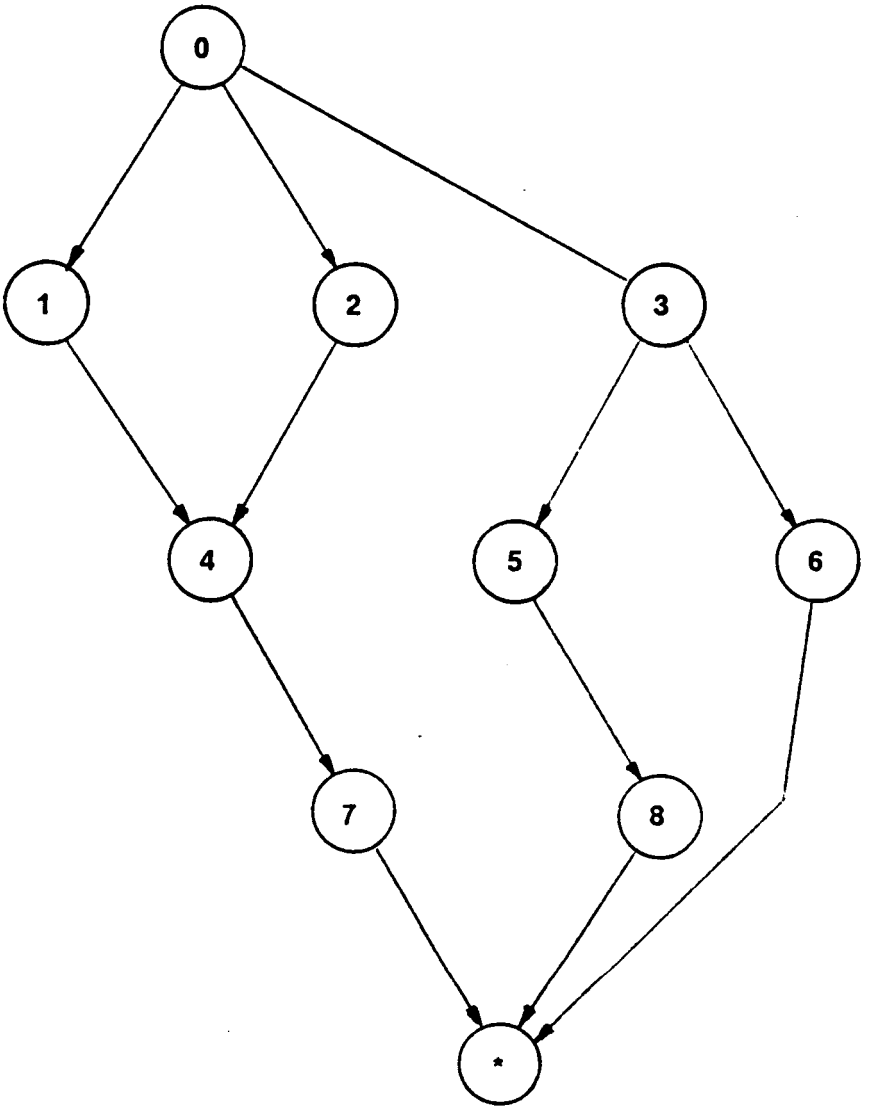


Figure 3.1 Example of a Task Graph

for the prototype problem under consideration. Hence it becomes a difficult task to identify not only the areas of mathematical parallelism [6] but also integrate these with solution techniques (like ABPC) under consideration.

A few important notions must be explicitly stated before any attempt is made to outline a systematic procedure for task graph development.

A "data flow graph" is very similar to a task graph except that the latter precludes all logical constructs of an incumbent program. In its simplest form, a task graph reflects an attempt to partition computation tasks in an optimum manner without any reference to logic statements which may have a representation in an equivalent data flow graph.

Being very closely related to the mathematical model of the system, a task graph is unique and specific to a particular application. The same system under different functional operations may require an entirely different task layout.

Even by partitioning the system model into several independent paths which may be computed in parallel, there exists a "critical path" which presents a set "lower limit" on the minimum achievable execution time. No amount of task decentralization in the form of a well balanced task graph or processor computing power can overcome the timing constraints set by the critical path. It is imperative that the update interval of data is greater than or atmost equal to the calculation time of the critical path.

3.2 Task Graph Development

A top-down design strategy is adopted in task graph development (see Figure 3.2). The system differential equations are partitioned and combined with standard integration techniques (ABPC in this case) to yield a set of difference equations. Subsequently, a data flow analysis is made wherein each difference equation is further broken up into simpler computation units in consonance with the mathematical attributes of the system. This procedure of task fragmentation is repeatedly continued till elementary computer operations (addition, subtraction, multiplication and division) or basic task units result. These are all interconnected and yield a complex mesh which is collectively called the "task graph" for the application under consideration. An attempt is made to keep the overall task graph reasonably balanced so as to preclude possibilities of unduly long critical paths.

To illustrate the above concepts, let us consider one of the differential equations having a high degree of cross coupling:

$$\dot{k}_{12}(t) = 20\ k_{12}(t)\ k_{22}(t) - k_{11}(t) + k_{12}(t) - 2\ k_{22}(t)$$

The first step is to make a data flow analysis for the equation above. This is done by constructing a function task block "f_{12}" (see figure 3.3). The nodes in the first level are either data constants or values of "k_{12}" and "k_{22}" at the previous update interval. The subsequent levels keep a numerical count of the elementary operations involved with "1*" within a node

TASK GRAPH DEVELOPMENT

DESIGN STRATEGY: TOP - DOWN APPROACH.

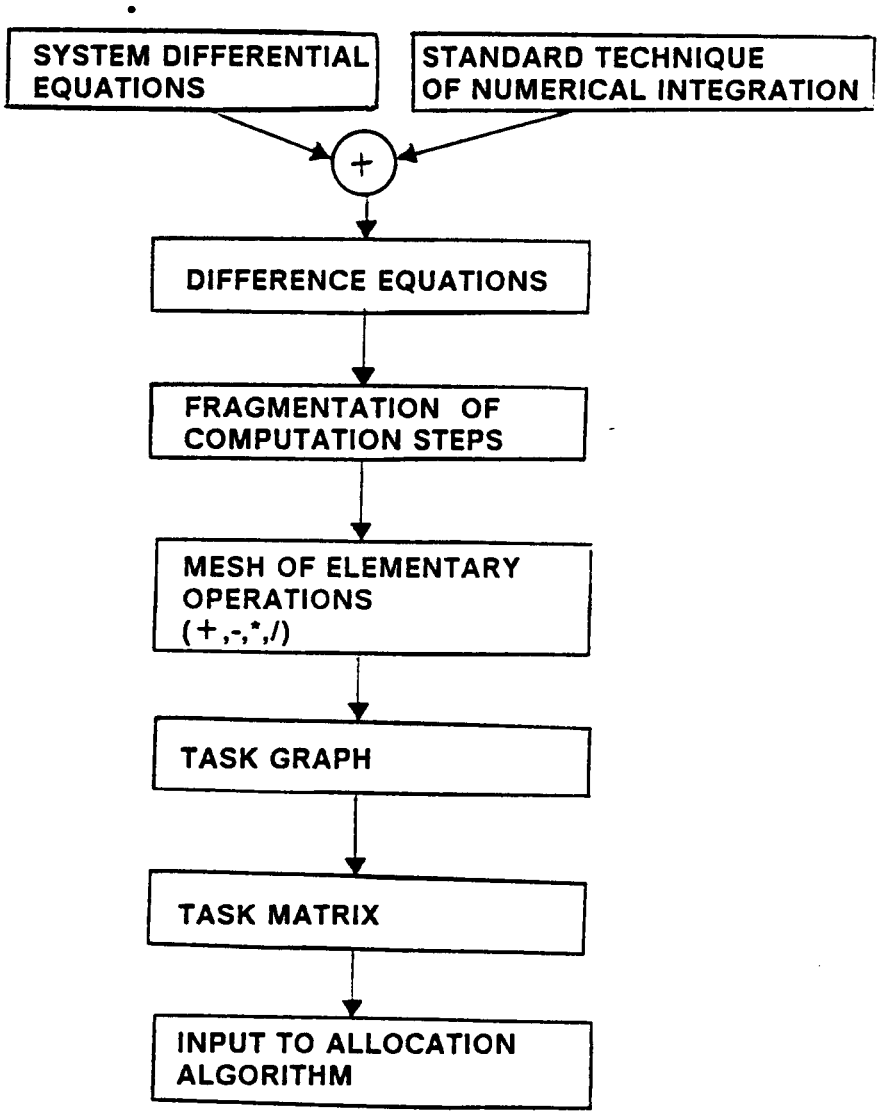


Figure 3.2 Task Graph Development

ORIGINAL PAGE IS
OF POOR QUALITY

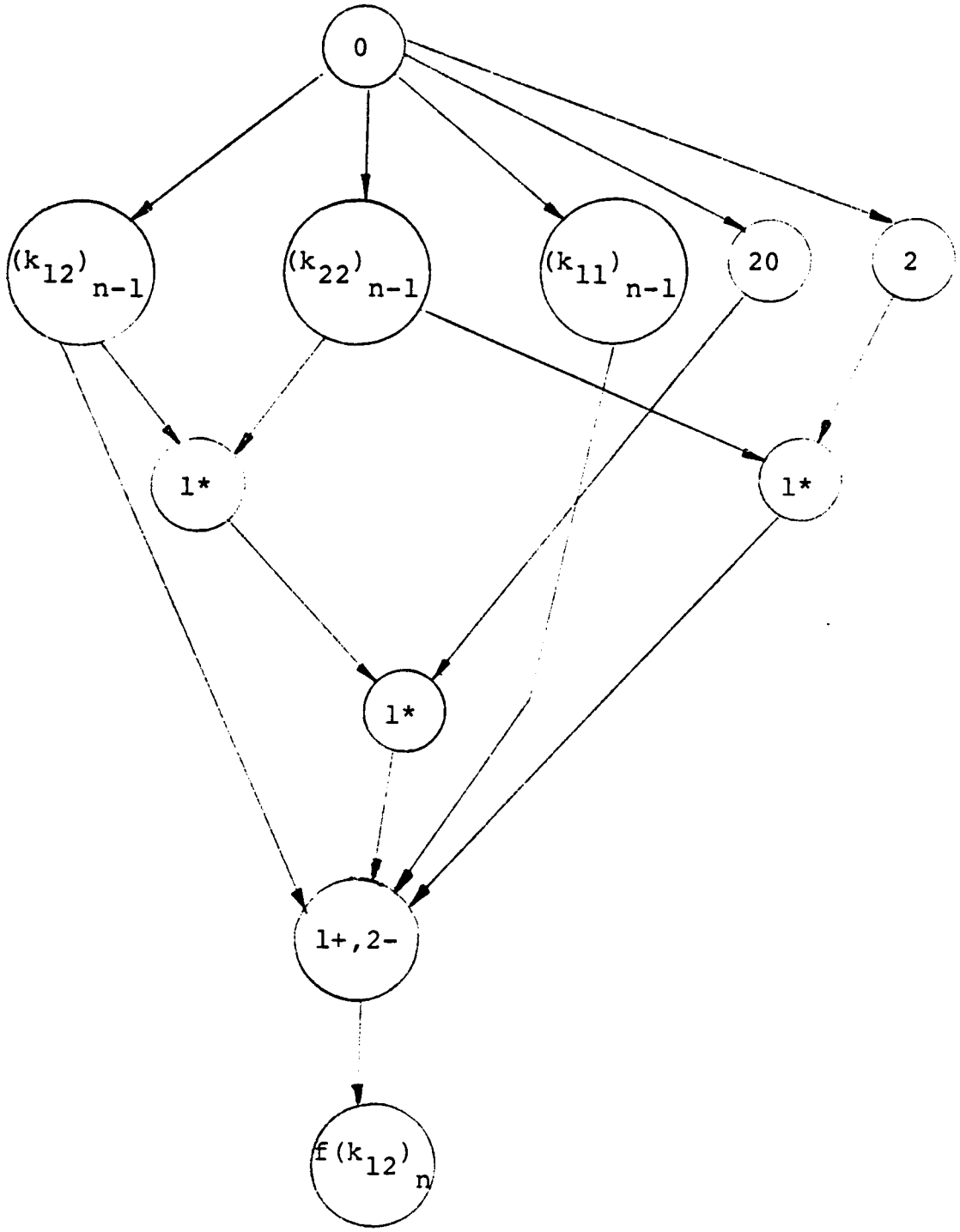


Figure 3.3 Function task block

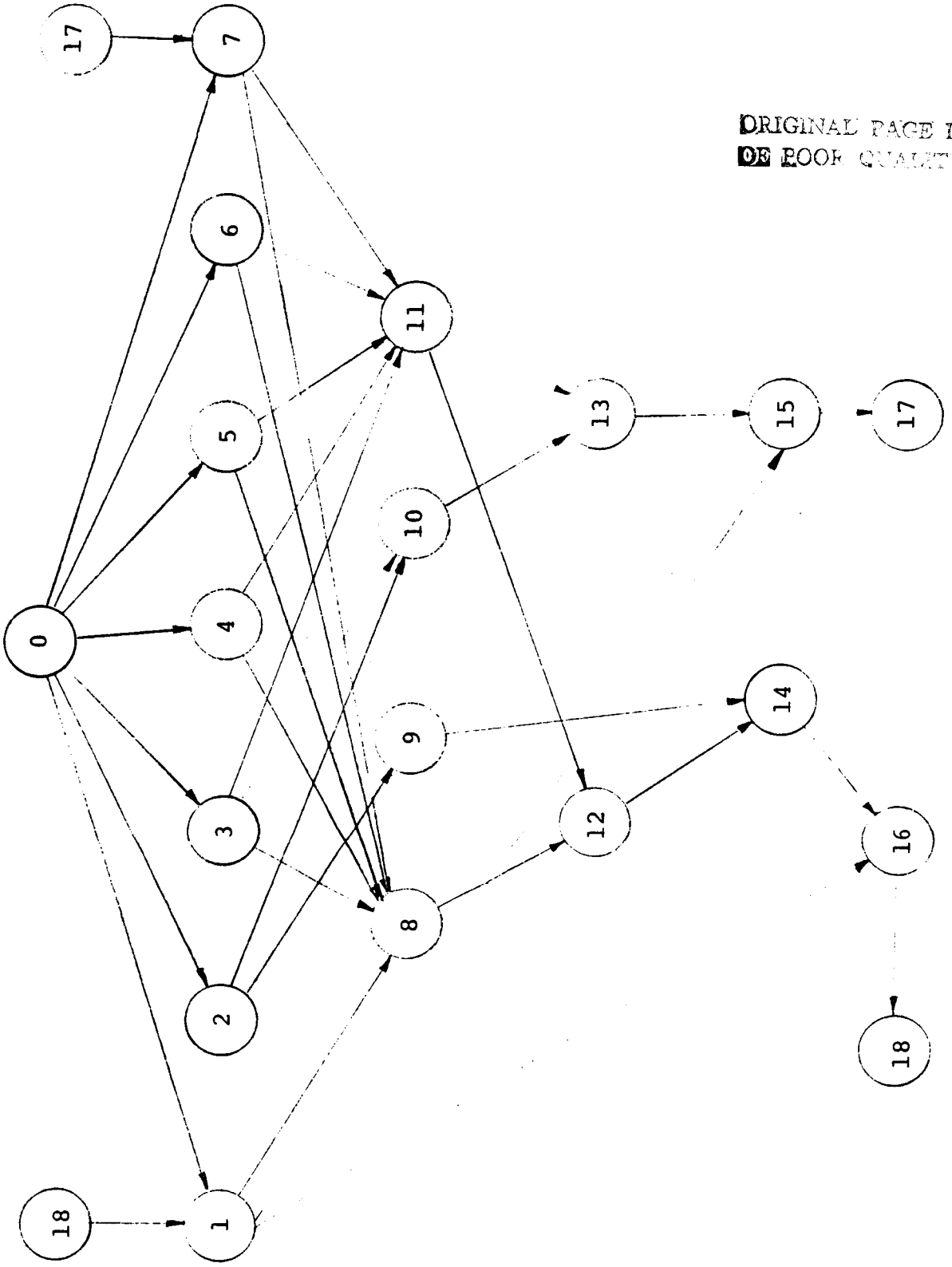
indicating one multiplication. Similarly, 1+,2- indicates a total of three operations comprising of one addition and two subtractions. Task time is counted on the basis of "time units" or TUs. Multiplication and division are assigned a weightage of 3 TUs compared to addition and subtraction which take 2 TUs. The function task block has a total count of 6 operations equaling at least 15 TUs.

The given equation along with the function task block must be integrated with the ABPC method. The difference equation to be solved becomes:

$$(k_{12})^{P}_{n-2} = (k_{12})^{C}_{n} - -2\ h\ f(k_{12})^{P}_{n-1}$$

$$(k_{12})^{C}_{n-1} = (k_{12})^{C}_{n} - h/2\ [\ f(k_{12})^{P}_{n-1} + f(k_{12})^{C}_{n}\ ]$$

Again on the basis of data flow, a track of the flow of computation is maintained and the resulting interconnected mesh of simple operations obtained constitutes the task graph for the equation in question (see figure 3.4). An interesting feature of this task graph is that it is non terminating in nature. Apart from the data constants, the parameter values are updated in every sampling interval. The systematic node description for the task graph under consideration is shown in Table 1. Each differential equation of the original set is thus fragmented to yield a sub task graph which are then interlinked to yield the overall task graph for the system. This has been shown in Appendix B.



ORIGINAL PAGE IS
OF POOR QUALITY

Figure 3.4 Task Graph for a Single System Equation

TABLE 1

NODE DESCRIPTION FOR TASK GRAPH IN FIGURE 3.4

Node No.	Parameter	Operation
1	$(k_{12})^c_n$	Load
2	h=constant	NOP
3	2=data constant	NOP
4	$(k_{22})^p_{n-1}$	Load
5	$(k_{11})^p_{n-1}$	Load
6	$(k_{12})^p_{n-1}$	Load
7	20=data constant	NOP
8	$f(k_{12})^c_n$	Load
9	---	/
10	---	*
11	$f(k_{12})^p_{n-1}$	Load
12	---	+
13	---	*
14	---	*
15	---	-
16	---	-
17	$(k_{12})^p_{n-2}$	Load
18	$(k_{12})^c_{n-1}$	Load

3.3 Task Matrix

A task graph for a practical problem is quite imposing in its complexity. A "Task Matrix" offers a convenient and concise technique for representing a task graph and at the same time maintains all precedence constraints. For a faithful representation, a task matrix should have the following fields:

1) Task Field (T): It indicates the task number.

2) Task Enable Field (E): It can assume only two values - a "HI" indicated by binary "1" and a "LO" indicated by a binary "0". Whenever E=1, the corresponding task is enabled.

3) Pending Task Queue Field (Q): It represents the number of tasks pending at each node. It provides a count of the immediate predecessor tasks that have to be executed prior to self execution. A task unit at a particular level in the task graph may be enabled only if the corresponding value of Q = 0.

4) Successor Field (S): This is in array field which keeps track of the number of immediate successor tasks at each node.

5) Weight Field (W): It shows the time taken for a task defined by the node under consideration to execute. The weight field is assigned arbitrarily as the speed of execution tends to vary with hardware features of the selected processor. However reasonable assumptions are made while assigning weights, e.g., task unit defining multiplication must have a larger execution time compared that which defines addition.

The task matrix table for the task graph in Figure 3.1 is shown (see Table 2). The tasks are numbered from "1" to "8" with

TABLE 2

TASK MATRIX FOR TASK GRAPH IN FIGURE 3.1

T	E	Q	S	W
1	1	0	4	X
2	1	0	4	X
3	1	0	5,6	X
4	0	2	7	X
5	0	1	8	X
6	0	1	•	X
7	0	1	•	X
8	0	1	•	X

T = TASK NUMBER FIELD.
E = TASK ENABLE FIELD.
Q = PENDING TASK QUEUE FIELD.
S = SUCCESSOR TASK FIELD.
W = WEIGHT FIELD.
X = DON'T CARE.

weights being "don't care" denoted by "X". "0" represents the input node whereas "*" denotes the terminal node. During start of execution any one of the tasks 1,2 and 3 may be executed and this is indicated by E = 1 and Q = 0 in corresponding fields. Task 4 has Q = 2 because it has two immediate pending or predecessor tasks in tasks 1 and 2. Tasks 5 and 6 are the successors of task 3 as shown in the S field. Tasks 6,7 and 8 terminate in the output node indicated by "*".

3.4 Scheduling Problem

The scheduling problem primarily deals with resource optimization. Stated simply it reduces to " Given a set of tasks or computations along with a set of operational precedence relationships that exist between a certain of these tasks, and given a set of `k' identical processors, how does one sequence or schedule these tasks on the `k' processors so that they execute in minimum time?" [8]. By definition a `scheduler' is an algorithm that uniquely specifies which job unit is to be serviced next by a resource [10] and to this end, an efficient scheduling algorithm need be developed which undertakes efficient task allocation and sequencing. Problems of this type are commonly referred to as "minimum execution time multiprocessor scheduling problem" [7].

3.5 Scheduling Classification

Task scheduling by itself forms an interesting area of research and draws heavily on concepts of graph theory and operations research. A number of scheduling strategies are in

vogue (see Table 3), each being suitable for a specific application. The major class of schedulers are categorized as pre-emptive or non pre-emptive.

A pre-emptive scheduler is capable of selecting and assigning a job to a server at any time irrespective of job completion, that is, a pre-emptive scheduler assumes that jobs are interruptible and will do so if another job of higher priority needs service. The overall flexibility of the schedule increases due to pre-emption but at the cost of hardware overhead and job "set-up" time. On the contrary, a non pre-emptive scheduler allows no job-switching, that is, once a job is assigned to a resource it has to be executed before another job can be accommodated even though it may have a higher priority.

3.6 Approaches to the Scheduling Algorithm

The scheduling problem may be approached from two different angles.

(1) Given a task graph and a set of `k' processors, a task assignment routine has to be developed that yields a description of the tasks done by each processor as a function of time. It ensures an optimum processor packing of task units so as to yield maximum resource utilization and at the same time attain a maximum speed of execution.

(2) Given a task graph, the scheduler keeps the option of available hardware open and selects an optimum number of processors for executing the task graph in minimum time. The

TABLE 3

SCHEDULING TECHNIQUES

Scheduler Name	Type of Operation
FCFS	First-come-first-served
SXFS	Shortest-job-first
LCFS	Least-completed-first
EDFS	Earliest-due-time-first
HSFS	Highest-static-priority-first
RR	Round robin

number of available processors in this case is a variable parameter which is optimally selected by the scheduling algorithm. This approach pre-supposes a flexible architecture for its realization since it needs a variable number of processors and sacrifices hardware utilization to get a higher throughput.

The scheduling algorithm that is developed is primarily based on the aforementioned second approach.

3.7 Assumptions in developing the Scheduling Algorithm

The scheduling algorithm developed is based on the following assumptions:

1) Scheduling is non pre-emptive and all task allocation is static.

2) Execution time of each task is known apriori.

3) Interprocessor and intraprocessor communication times are negligible.

4) Task weights are assigned arbitrarily but uniformity is maintained between comparable tasks. Tasks requiring longer CPU time (like multiplication) have been assigned larger weights compared to tasks requiring lower CPU time (like register move, addition etc.). Such arbitrariness is primarily due to lack of well defined execution-time standards on account of the widely varying processor types available currently. Moreover, conceptually the algorithmic implementation is independent of the weights assigned to the task units.

3.8 Scheduling Algorithm

The scheduling algorithm (originally credited to Oschner) maps the task graph onto a task matrix and seeks to obtain an optimum schedule by means of elementary operations on the task matrix. The step by step detail for the algorithm is as follows:

1) A task matrix is defined by five fields T,E,Q,S,W.

2) A task is enabled only when E=1 and Q=0

3) An enabled task can be allocated to a free PE only.

4) A task unit assigned to a PE has its E field decremented to zero, that is, E=0 for an assigned task unit.

5) After task completion, the successor or S field of the task is examined so as to decrement the Q field of each successor.

6) All successor tasks having Q=0 as a result of decrement are enabled.

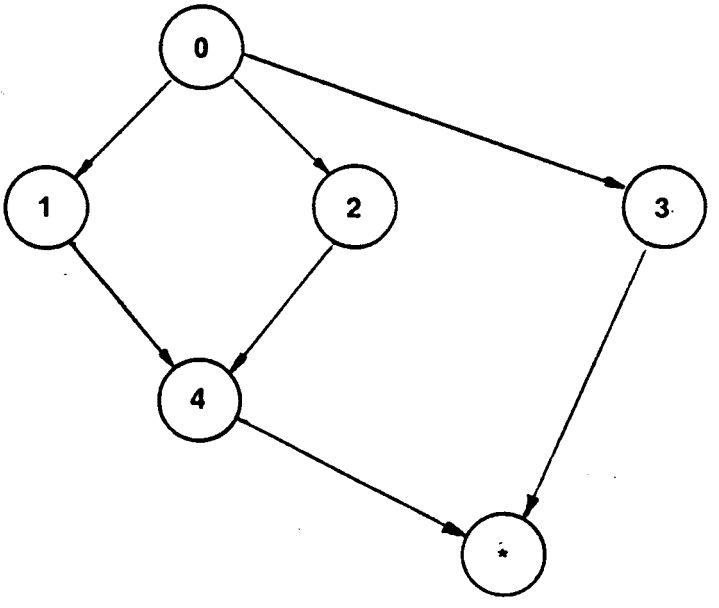
7) Repeated execution whenever a PE becomes idle.

8) Scheduling is complete when all tasks have E=0 and Q=0.

As a specific example, a simple task graph and associated task matrix is considered (see Table 4). Initially any one of tasks 1, 2 and 3 may be allocated depending on the number of processors available. Assuming that all tasks are assigned, execution (time_processing in Pascal routine - Appendix D) begins and the respective "E" fields are reduced to zero (see Table 5). Task 1 having minimum weight is completed first so that the PE to which it is assigned is the first to become idle.

TABLE 4

TASK GRAPH AND TASK MATRIX



T	E	Q	S	W
1	1	0	4	10
2	1	0	4	20
3	1	0	*	30
4	0	2	*	10

TABLE 5

ELEMENTARY OPERATION ON TASK MATRIX

T	E	Q	S	W
1	0	0	4	10
2	0	0	4	20
3	0	0	•	30
4	0	2	•	10

TABLE 6

ELEMENTARY OPERATION ON TASK MATRIX

T	E	Q	S	W
1	0	0	4	10
2	0	0	4	20
3	0	0	•	30
4	0	1	•	10

When this stage is reached, the scheduling process takes over. The successor field of task 1 is examined which points to task 4. The scheduler now decrements the Q field of task 4 thereby making it equal to 1 (see table 6).

Even though task 1 is complete, task 4 cannot be assigned until task 2 ends. So task execution starts again with PE to which task 1 was assigned remaining idle. When task 2 is completed, the scheduler looks at the corresponding S field which again points to task 4. The Q field of task 4 is decremented to zero as a result. The scheduler now sets the E field of task 4 thereby enabling it (see Table 7). Task 4 is assigned to an available PE and its E field is reduced to zero. When all tasks have been assigned and execution is complete, the E and Q fields of all tasks equal zero and the resulting task matrix is shown in Table 8.

From this example, it becomes clear that by elementary operations (like look up, decrement etc.) it is possible to keep a dynamic track of a variable number of tasks and PEs. The resulting information is adequate to set up a timing diagram or "Gantt Chart" schedule for each PE which is of considerable help in calculating the overall time necessary to execute the task graph. By the varying the number of processors used, considerable insight on overall performance is obtained. These factors are discussed subsequently.

TABLE 7

ELEMENTARY OPERATION ON TASK MATRIX

T	E	Q	S	W
1	0	0	4	10
2	0	0	4	20
3	0	0	•	30
4	1	0	•	10

TABLE 8

ELEMENTARY OPERATION ON TASK MATRIX

T	E	Q	S	W
1	0	0	4	10
2	0	0	4	20
3	0	0	*	30
4	0	0	*	10

CHAPTER 4

SIMULATION AND PERFORMANCE EVALUATION

The evaluation of a computer system generally involves the following classes of considerations:

1) Performance
2) Cost
3) User convenience
4) Reliability

An attempt is made here to provide a critical appraisal of overall performance improvement when the system under consideration is subjected to the previously described parallel model of implementation.

4.1 Performance Evaluation Criterion

The primary requirements for performance evaluation are:

1) Analysis
2) Simulation
3) Measurements

Analysis and simulation is accomplished by partitioning the system differential equations into task units which are then allocated to a variable set of processors. The merit of the scheme is judged on the basis of the following performance indices:

1) Execution time

2) Percentage speed-up

3) Percentage efficiency

Execution time may be defined as the time required by a given set of processors to execute the task graph in question. For a real-time control problem, the execution time is of great significance and must be less than the periodic update time.

The increase in speed of computation with a larger number of processors compared to that of an uniprocessor is generally denoted by the percentage speed-up factor. If "t" is the time required to execute a task graph using a set of "p" processors and "m" equals the time to do the same using a single processor, then speed-up factor [9] is given by:

$$\text{speed-up} = (m / t)$$

The percentage efficiency shows the overall resource utilization for a parallel implementation. Mathematically,

$$\% \text{ efficiency} = (m / tp) * 100$$

Percentage efficiency is a measure of the idle time of the PEs. It has a value of 100% for an uniprocessor system as can be verified from the mathematical expression.

4.2 Assumptions in Simulation

To facilitate and simplify analysis, the following model for a parallel implementation is adopted:

1) an unlimited number of processors is available.

2) each PE is capable of evaluating any of the four fundamental arithmetic operations (+, -, *, /).

3) data and memory alignment times are neglected.

Although assumptions 1) and 3) appear unrealistic, decreasing hardware costs are giving rise to large multiprocessor systems which have almost an unlimited number of processors , eg., The Hypercube, The Butterfly Computer which has 256 PEs with scope for further expansion. Similarly, data and memory time penalties simply offset the computation results by a fixed factor and therefore do not form a barrier to the conceptual implementation of a parallel model.

4.3 Results of Simulation

The task flow pattern for the linear system is simulated using a variable number of PEs and at each stage the aforementioned performance indices are recorded. A graphical representation of these indicate interesting highlights .

The execution time curve (see Figure 4.1) droops sharply as the number of processors increase showing that with increase in the number of PEs the task completion time rapidly decreases. The curve has a characteristic hump in the vicinity of ten PEs. Any further attempt to boost computing power by increasing the number of PEs has negligible effect thereby indicating that time corresponding to critical path has been reached.

The percentage efficiency curve (see figure 4.2) initially remains at a high value which implies that available tasks are adequate to keep the set of processors occupied throughout the

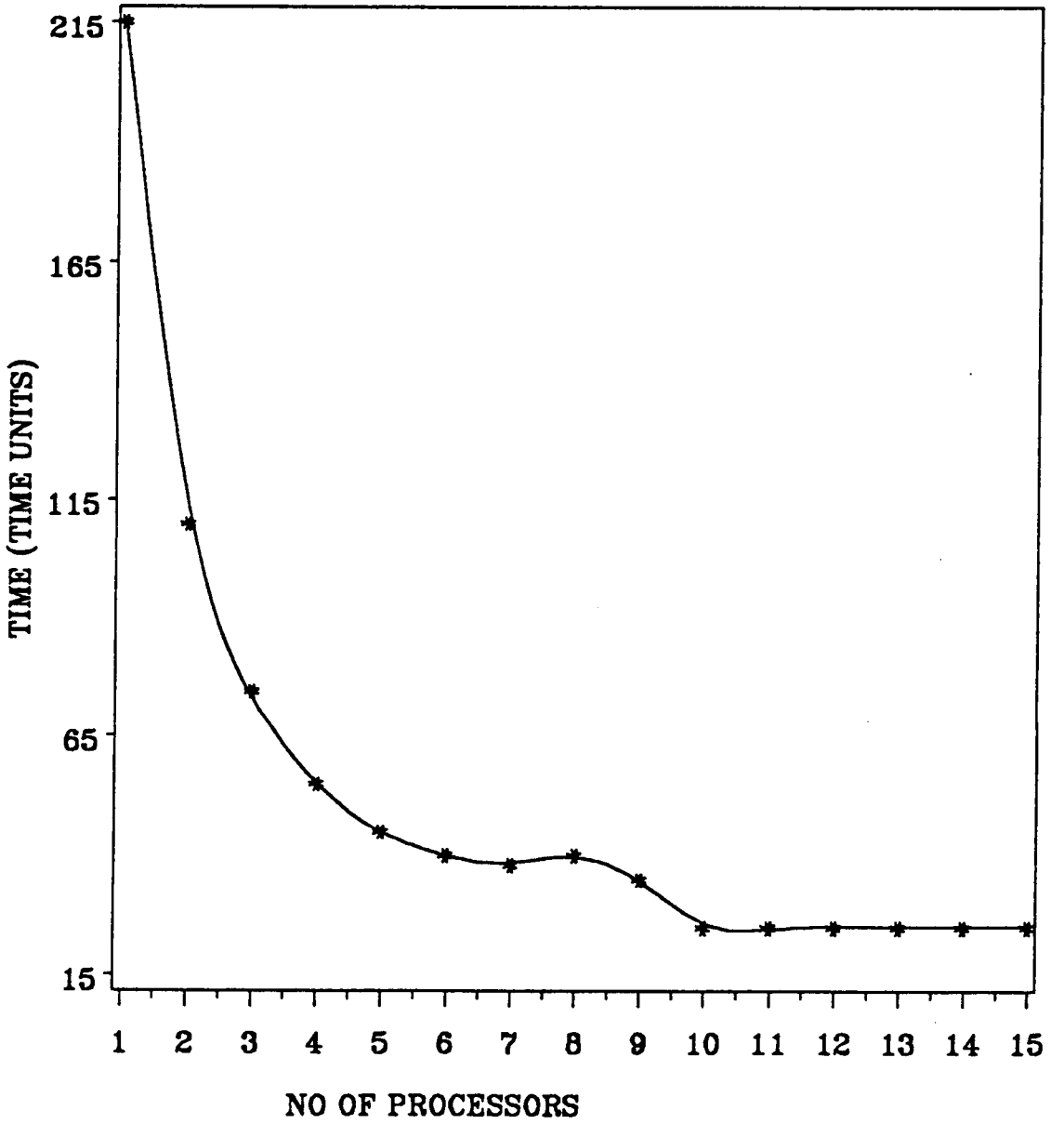


Figure 4.1 Processor Execution Time

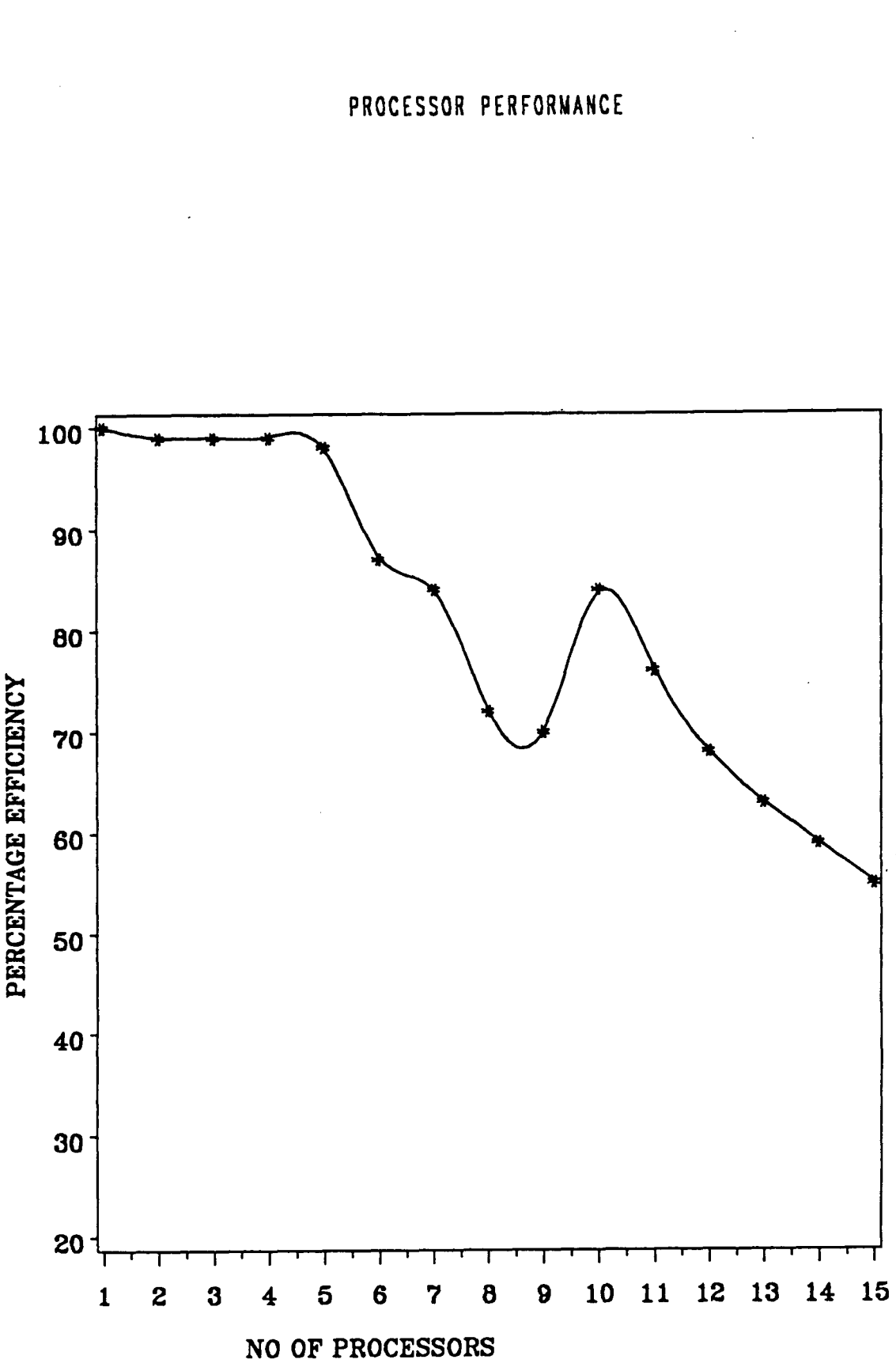


Figure 4.2 Processor Efficiency

update interval. However, for more than five PEs it rapidly decreases owing to the idle time generated. This trend continues till for ten PEs the curve has a local maxima corresponding to a percentage efficiency of approximately 85%. Beyond this, the efficiency curve again toggles down. The logical inference drawn is that for a set of ten PEs a compromise is affected between idle time and speed of execution whereby resource efficiency is sacrificed to obtain a greater speed advantage. This is also corroborated by the speed up curve (see Figure 4.3) which indicates that beyond ten PEs the speed up ratio remains unaltered. The performance indices therefore point to ten PEs as an optimum selection for the task graph under consideration. The task allocation scheme for the optimum number of PEs is generated as output by the scheduling program. A Gantt Chart or a processor timing diagram can be set up from the results. It may be noted that a close processor packing of tasks exist and overall idle time is negligible. The task graph, task matrix, program output and Gantt chart are listed in Appendix B.

PROCESSOR PERFORMANCE

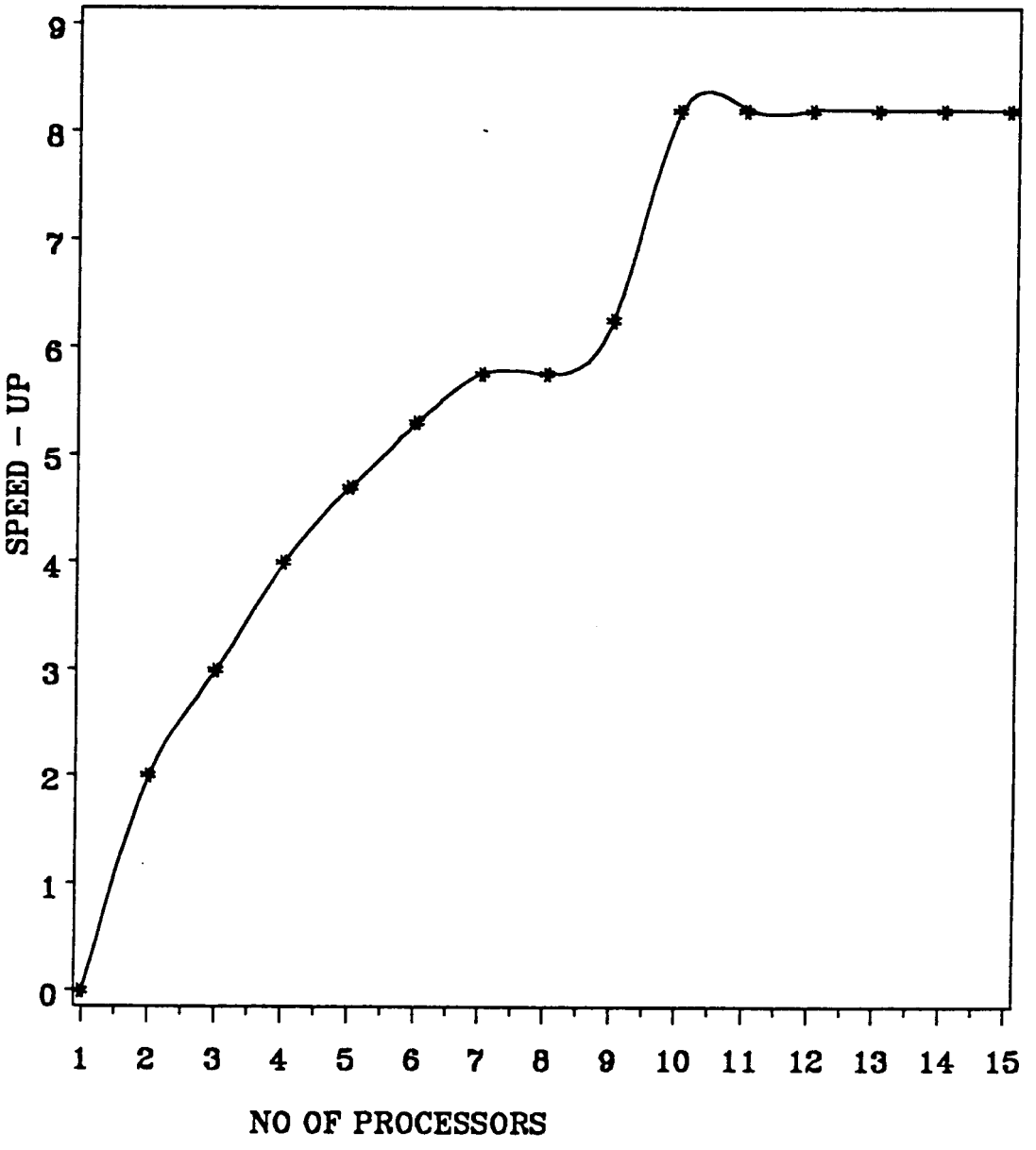


Figure 4.3 Processor Speed-Up

CHAPTER 5

ARCHITECTURE AND HARDWARE DESIGN

Conventional computers solve problems one step at a time. Advanced parallel computers are able to execute independent parts of the problem concurrently thereby reducing overall execution time [13]. The success of a parallel implementation depends entirely on the hardware support and to this end an efficient architecture is proposed.

5.1 Architectural Requirements

Computer architecture encompasses a very wide area of knowledge bounded by ever changing innovations. It is extremely difficult to define all attributes necessary to justify a particular architecture. In this thesis research, a multiprocessor parallel algorithmic implementation has been proposed which in turn needs a truly parallel hardware back up.

Flexibility is one of most desirable features for such an architecture. A task graph corresponds uniquely to an application . Any changes in application demands a new task graph which in turn requires an altered hardware support. Hence, a truly parallel machine must have hardware upgradability and reconfigurability. Popular parallel machines like the Butterfly Computer, Hypercube, REMPS [14, 15] etc. incorporate

this philosophy. Current researches on the FAST at the University of Alabama also re-emphasizes this point.

The PE system architecture must have a high degree of pipelining to reduce intermediate idle time. It is also imperative for each PE to have an on-chip in addition to global memory. This reduces the conventional "Von Neumann" bottleneck and increases computing power.

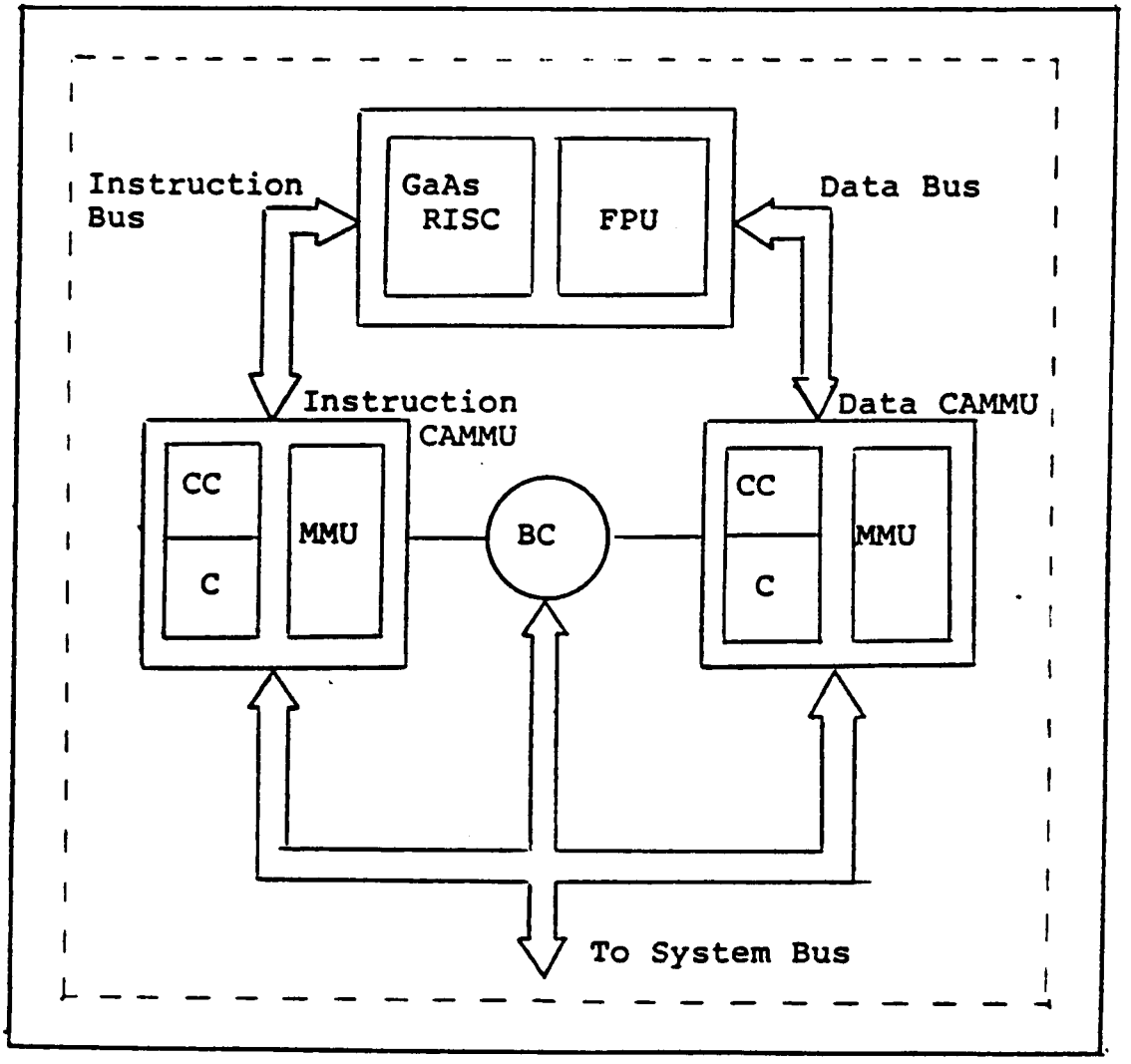
5.2 PE System Design

A large number of PEs with excellent functional features are currently available [16, 17]. However, a futuristic PE design is proposed here (see Figure 5.1). A gallium arsenide RISC engine is coupled with a floating point coprocessor unit and constitutes the core of the processing element [18, 19]. These are connected by instruction and data buses to respective caches which virtually eliminates all global memory accesses except perhaps at the pre-processing stage [20]. Separate instruction and data caches reduce cache-contention and internal bus traffic. The PE interfaces with the system bus using a bus controller.

5.3 Technology Selection

An ambitious proposition using WSI GaAs is recommended. Although a great majority of the integrated circuits are fabricated with silicon, GaAs technology offers several advantages [20]:

1) GaAs chips are five to ten times faster than fastest silicon chips.



FPU = Floating Point Unit
CC = Cache Controller
MMU = Memory Management Unit
BC = Bus Controller
CAMMU = Cache and Memory Management Unit

Figure 5.1 PE Design Schemata

2) It is radiation "hard" and operates over a wide temperature range (-200^0C to $+200^0$C).

3) It is also better suited for efficient integration with electronic and optical components.

Although high cost and low levels of integration are major drawbacks, these are expected to be eliminated as the technology matures.

Wafer-scale-integration denotes the level of integration attained when an entire wafer is used is used to fabricate a circuit. Currently WSI is the highest level of integration for monolithic circuits [21]. The technology is still plagued by problems of heat dissipation and low production yield. However, higher attainable density levels and fewer off chip connections are major factors in proposing this futuristic technology that has already started making inroads in the chip market [22].

5.4 Interconnection and System Layout

A hierarchical fiber optic star (see Figure 5.2) is proposed as a suitable system layout and corresponds to the FAST architecture [23]. Such a structure is easily expandable and provides an inexhaustible source of computing power. Each tentacle of the star ends in individual processing modules which may be specialized to perform functions like error checking, I/O, communication, numeric processing etc. Such a system has the option of having heterogeneous modules or homogeneous modules depending upon the application. Each fiber optic star cluster may be configured to form specialized hardware modules

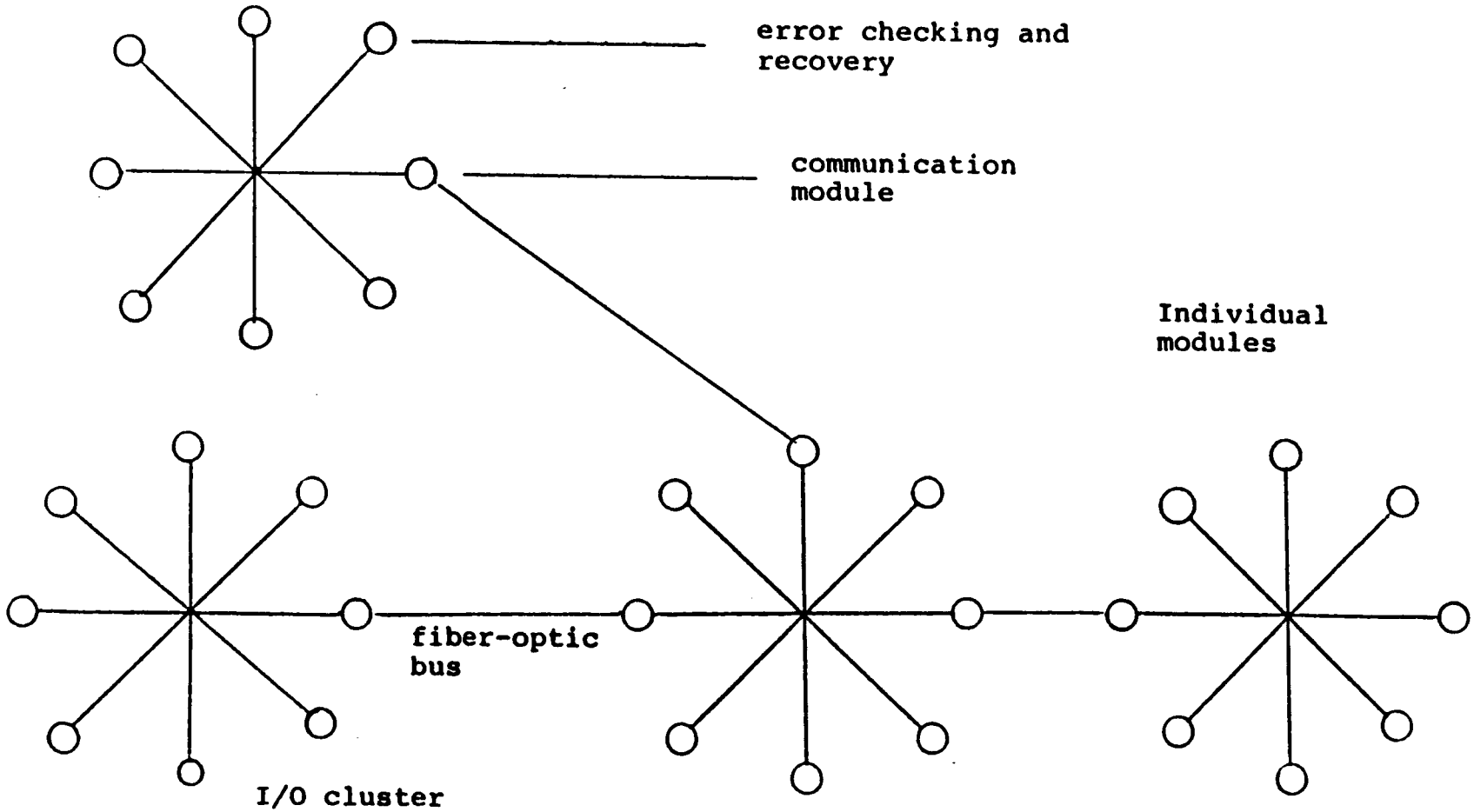


Figure 5.2 System Architecture Layout

for efficient task execution. Optical fiber communication links are optimally compatible with GaAs WSI technology and is sufficient to meet the highest transfer rates [24].

5.5 Future Directions

Although a futuristic hardware support is proposed, architectural innovations may still be implemented to attain higher modularity and efficiency. Considerable work needs to be done in the development of parallel software bases which still happens to be inherently sequential [25]. The setting up of a task graph for different applications is wasteful of manhours. Automated software packages need to be developed for performing domain and functional decomposition. The future will undoubtedly be affected by improvements in semiconductor technology. However, any drastic performance improvement would need a technological breakthrough, like the development of high temperature superconductors etc., but the basic tenets of parallel processing are going to hold good for some time to come.

REFERENCES

[1] G. C. Fox and P. C. Messina, "Advanced Computer Architectures", Scientific American, vol 257, pp. 67-75, October 1987.

[2] D. Peng and K. G. Shin, "Modelling of Concurrent Task Execution in a Distributed System for Real-Time Control", IEEE Transactions on Computers, vol. c36, no.4, pp. 500-516, April 1987.

[3] Optimal Control Theory - An Introduction ; by Donald E. Kirk; Prentice-Hall; 1970.

[4] L. G. Birta and O. Abou Rabia, " Parallel Block Predictor-Corrector Methods for ODES", IEEE Transactions on Computers, vol. c36, no.4, pp. 299-311, March 1987.

[5] W. L. Miranker and W. Liniger, "Parallel Methods for the Numerical Integration of Ordinary Differential Equations", Math. Comput., vol. 21, pp. 303-320, Nov. 1967

[6] D. J. Arpasi and E. J. Milner, "Mathematical Model Partitioning and Packing for Parallel Computer Calculation", pp. 67-74, NASA TM-87170

[7] H. Kasahara and S. Narita, "Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing", IEEE Transactions on Computers, vol. c33, no.11, pp. 1023-1029, Nov. 1984

[8] R. R. Muntz and E. G. Coffman, Jr., "Optimal Premptive Scheduling on Two - Processor Systems", IEEE Transactions on Computers, vol. c18, no.11, pp. 1014 -1020, Nov. 1969.

[9] C. V. Ramamoorthy, K. M. Chandy, and M. J. Gonzalez, "Optimal Scheduling Strategies in a Multiprocessor System", IEEE Transactions on Computers, vol. c21, no.2, pp. 137-146, Feb. 1972.

[10] Introduction to the Design and Analysis of Algorithms; by S. E. Goodman and S. T. Hedetniemi; McGraw Hill Book Company; 1977.

[11] Computer System Performance; by H. Hellerman and T. F. Conroy; McGraw Hill Book Company; 1975.

[12] A. H. Sameh, " Numerical Parallel Algorithms - A Survey", High Speed Computer and Organization, pp. 207-228, 1977.

[13] R. Cytron, " Useful Parallelism in Multiprocessing Environment", Proceedings of the 1985 International Conference on Parallel Processing, pp. 450-457.

[14] K. Hwang and Z. Xu, " REMPS: A Reconfigurable Multiprocessor for Scientific Supercomputing", Proceedings of the 1985 International Conference on Parallel Processing, pp. 102-111.

[15] J. C. Peterson, J. O. Tuazon, D. Lieberman and M. Pniel, "The Mark III Hypercube-Ensemble Concurrent Computer", Proceedings of the 1985 International Conference on Parallel Processing, pp. 71-73.

[16] R. P. Bianchini and J. P. Shen, "Interprocessor Traffic Scheduling Algorithms for Multiple-Processor Networks", IEEE Transactions on Computers, vol. c36, no.4, pp. 396-409, April 1987.

[17] T. L. Johnson, "The RISC/CISC Melting Pot", BYTE, pp. 153-160, April 1987.

[18] J. F. Mcdonald, H. J. Greub, R. H. Steinworth, B. J. Donlan and A. S. Bergendahl, "Wafer Scale Interconnections for GaAs Packaging - Application to RISC Architecture", IEEE Computer, pp. 21-34, April 1987.

[19] V. Milutinovic, "An Introduction to GaAs microprocessor architecture for VLSI", IEEE Computer, pp. 30-42, March 1986.

[20] V. Milutinovic, " GaAs Microprocessor Technology", IEEE Computer", pp. 10-13, October 1986.

[21] J. F. Mcdonald, " The Trials of Wafer-Scale Integration", IEEE Spectrum, pp. 32-39, October 1984.

[22] R. O. Carlson, "Future trends in Wafer-Scale Integration", Proceedings of the IEEE, pp. 1741-1752, December 1986.

[23] L. D. Huthceson, "Optical interconnects replace hardwire", IEEE Spectrum, pp. 30-35, March 1987.

[24] D. H. Hartman, "An effective lateral fiber-optic electronic coupling and packaging technique suitable for VHSIC applications, Journal of Lightwave Technology, pp. 73-81, Jan 1986.

[25] A. H. Karp, "Programming for Parallelism", IEEE Computer, pp. 43-56, May 1987.

APPENDIX A

SOLUTION METHOD FOR OPTIMAL CONTROL PROBLEMS USING MATRIX RICATTI EQUATIONS

Several techniques are available for the solution of optimal control problems. A widely used method involves the setting up of Matrix Ricatti equations.

The state equations are :

$$\dot{x}(t) = A(t)x(t) + B(t)u(t)$$

and the performance measure to be minimised is

$$J = 0.5[x(t_f) - r(t_f)]^T H[x(t_f) - r(t_f)] + 0.5 \int_{t_o}^{t_f} \{[x(t) - r(t)]^T Q(t)[x(t) - r(t)] + u^T(t)R(t)u(t)\}\, dt$$

where r(t) is the desired value of the state vector. H and Q are positive semidefinite matrices, and R is real symmetric and positive definite. The final time "t_f" is fixed.

The Hamiltonian is given by

$$h(x(t),u(t),p(t),t) = 0.5 \|x(t) - r(t)\|^2_{Q(t)}$$

$$\|u(t)\|^2_{R(t)} + p^T(t)A(t)x(t) + p^T(t)B(t)u(t)$$

The costate equations are

$$\dot{p}^*(t) = - Q(t)x^*(t) - A^T(t)p^*(t) + Q(t)r(t)$$

and the algebraic relations to be satisfied are

$$0 = R(t)u^*(t) + B^T(t)p^*(t)$$

This yields the optimal control law in terms of the costate equation as

$$u^*(t) = -R^{-1}(t)B^T(t)p^*(t)$$

Instead of computing the STM, an easier computational alternative is to express

$$p^*(t) = K(t)x^*(t) + s(t)$$

Differentiating both sides with respect to "t", we get

$$\dot{p}^*(t) = \dot{K}(t)x^*(t) + K(t)\dot{x}^*(t) + \dot{s}(t)$$

Substituting for $\dot{p}^*(t)$ and $\dot{x}^*(t)$ and then eliminating $p^*(t)$, the following equations, commonly referred to as the Matrix Ricatti equations, are obtained

$$\dot{K}(t)= -K(t)A(t) - A^T(t)K(t) - Q(t) + K(t)B(t)R^{-1}(t)B^T(t)K(t)$$

and

$$\dot{s}(t) = -[A^T(t) - K(t)B(t)R^{-1}(t)B^T(t)]s(t) + Q(t)r(t)$$

"K" is a symmetric matrix of order "n" by "n" and "s" is a "n" by 1 vector. Hence a set of "[n{n+1}/2]+n" first-order differential equations need to solved. The boundary conditions are

$$p^*(t_f) = Hx^*(t_f) - Hr(t_f)$$
$$= K(t_f)x^*(t_f) + s(t_f)$$

As all $x^*(t_f)$ and $r(t_f)$ satisfy these equations, the boundary conditions are

$$K(t_f) = H$$

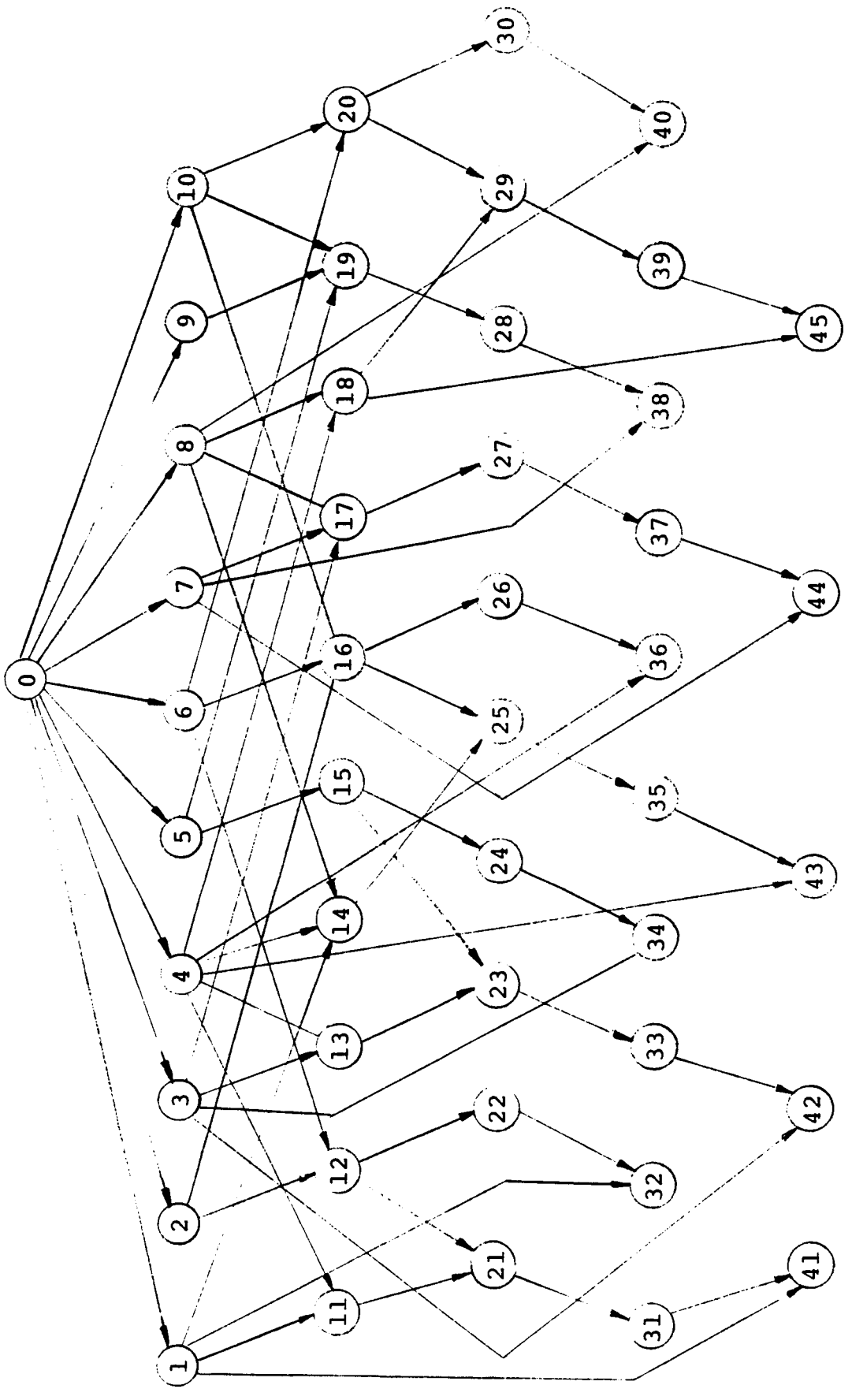
and

$$s(t_f) = -Hr(t_f)$$

The optimal control law may be computed from the values of "K" and "s" by means of standard integration techniques.

APPENDIX B

TASK GRAPH ATTRIBUTES FOR HIGHLY-COUPLED LINEAR SYSTEM EQUATIONS



OVERALL SYSTEM TASK GRAPH

Node No.	Parameter	Operation
1	$(k_{11})^{c}_{n}$	Load
2	$(k_{11})^{p}_{n-1}$	Load
3	$(s_1)^{c}_{n}$	Load
4	$(k_{12})^{c}_{n}$	Load
5	$(s_1)^{p}_{n-1}$	Load
6	$(k_{12})^{p}_{n-1}$	Load
7	$(s_2)^{c}_{n}$	Load
8	$(k_{22})^{c}_{n}$	Load
9	$(s_2)^{p}_{n-1}$	Load
10	$(k_{22})^{p}_{n-1}$	Load
11	$f(k_{11})^{c}_{n}$	function subtask
12	$f(k_{11})^{p}_{n-1}$	"
13	$f(s_1)^{c}_{n}$	"
14	$f(k_{12})^{c}_{n}$	"
15	$f(s_1)^{p}_{n-1}$	"
16	$f(k_{12})^{p}_{n-1}$	"
17	$f(s_2)^{c}_{n}$	"
18	$f(k_{22})^{c}_{n}$	"
19	$f(s_2)^{p}_{n-1}$	"
20	$f(k_{22})^{p}_{n-1}$	"

Node Description for System Task Graph

Node no.	Parameter	Operation
21	---	+
22	---	*
23	---	+
24	---	*
25	---	+
26	---	*
27	---	+
28	---	*
29	---	+
30	---	*
31	---	*
32	---	-
33	---	*
34	---	-
35	---	*
36	---	-
37	---	*
38	---	-
39	---	*
40	---	-
41	---	-
42	---	-
43	---	-
44	---	-
45	---	-

Node Description for System Task Graph

DATA BASE FOR TASK MATRIX

45

1
1
1
1
1
1
1
1
1
1
1
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0
0

0
0
0
0
0
0
0
0
0
0
2
2
2
4
1
3
4
2
3
2
2
1
2
1
2
1
2
1
2
1
1
2
1
2
1
2
1
2
1
2
2
2
2
2
2
4

11
14
32
41
0
0
2
12
16
0
0
0
0
4
13
17
34
42
0
0
6
11
13
14
18
36
43
2
15
19
0
0
0
0
3
12
16
20
0
0
0
3
17
38
44
0

0
0
4
14
17
18
40
0
0
1
19
0
0
0
0
0
3
16
19
20
0
0
0
1
21
0
0
0
0
0
2
21
22
0
0
0
0
1
23
0
0
0
0
0
1
25

0
0
0
0
0
2
23
24
0
0
0
0
2
25
26
0
0
0
0
1
27
0
0
0
0
0
2
29
45
0
0
0
0
2
27
28
0
0
0
0
2
29
30
0
0
0

0
1
31
0
0
0
0
0
1
32
0
0
0
0
0
1
33
0
0
0
0
0
1
34
0
0
0
0
0
1
35
0
0
0
0
0
1
36
0
0
0
0
0
1
37
0

0
0
0
0
1
38
0
0
0
0
0
1
39
0
0
0
0
0
1
40
0
0
0
0
0
1
41
0
0
0
0
0
1
0
0
0
0
0
0
1
42
0
0
0
0
0

1
0
0
0
0
0
0
1
43
0
0
0
0
0
1
0
0
0
0
0
0
0
1
44
0
0
0
0
0
1
0
0
0
0
0
0
0
1
45
0
0
0
0
0
1
0
0

0
0
0
0
1
0
0
0
0
0
0
1
0
0
0
0
0
0
1
0
0
0
0
0
0
0
1
0
0
0
0
0
0
0
1
1
1
1
1
1
1

1
1
1
13
13
18
15
18
15
10
16
10
16
2
3
2
3
2
3
2
3
2
3
3
2
3
2
3
2
3
2
3
2
2
2
2
2

TASK ALLOCATION

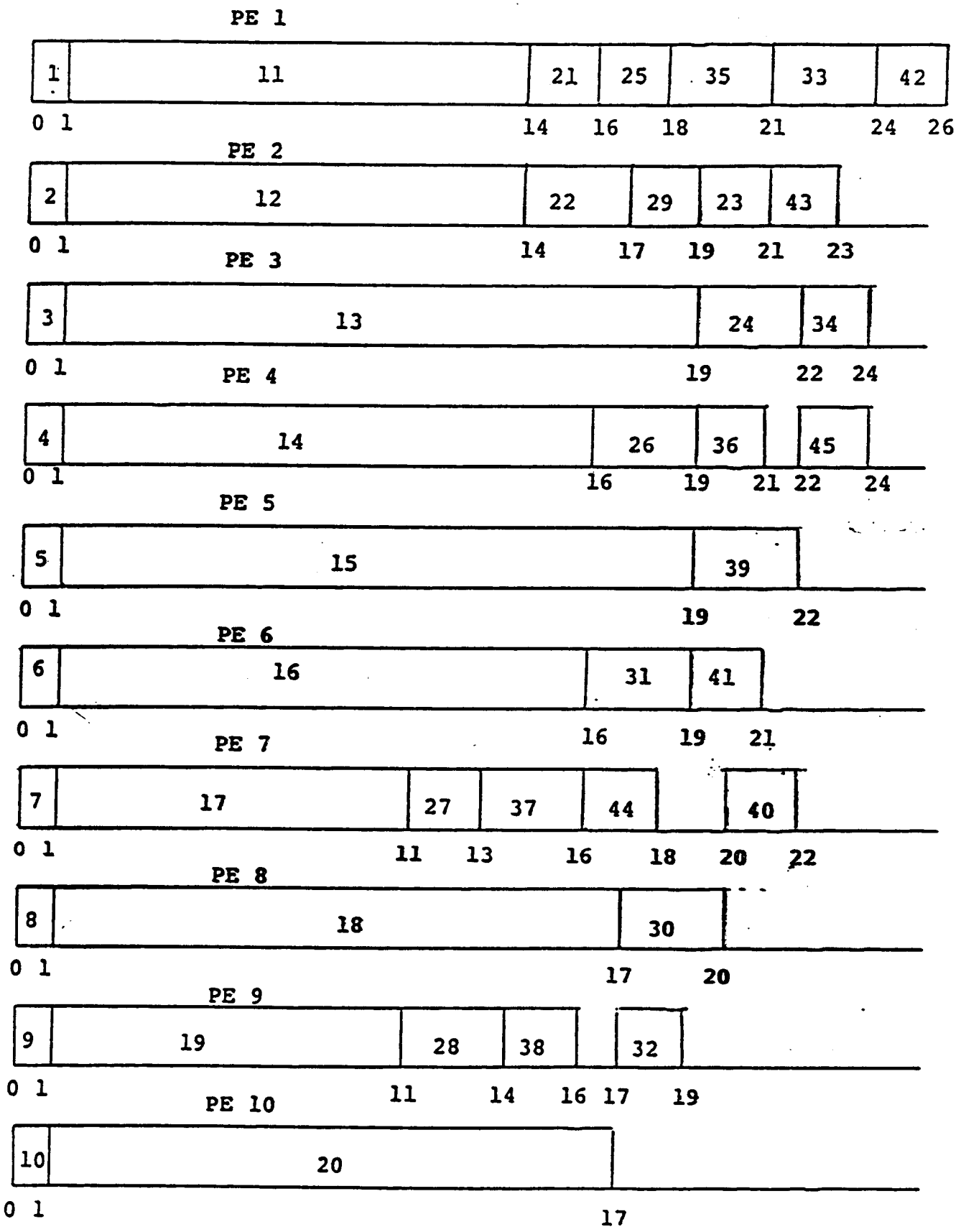
THE NUMBER OF PROCESSORS USED=10

THE NUMBER OF DEFINED TASKS=45

processor [1] assigned task [1]
processor [2] assigned task [2]
processor [3] assigned task [3]
processor [4] assigned task [4]
processor [5] assigned task [5]
processor [6] assigned task [6]
processor [7] assigned task [7]
processor [8] assigned task [8]
processor [9] assigned task [9]
processor [10] assigned task [10]
processor [1] assigned task [11]
processor [2] assigned task [12]
processor [3] assigned task [13]
processor [4] assigned task [14]
processor [5] assigned task [15]
processor [6] assigned task [16]
processor [7] assigned task [17]
processor [8] assigned task [18]
processor [9] assigned task [19]
processor [10] assigned task [20]
processor [7] assigned task [27]
processor [9] assigned task [28]
processor [7] assigned task [37]
processor [1] assigned task [21]
processor [2] assigned task [22]
processor [9] assigned task [38]
processor [1] assigned task [25]
processor [4] assigned task [26]
processor [6] assigned task [31]
processor [7] assigned task [44]
processor number [9] idle for 1 TUS
processor [2] assigned task [29]
processor [8] assigned task [30]
processor [9] assigned task [32]
processor number [10] idle for 1 TUS
processor [1] assigned task [35]
processor number [7] idle for 1 TUS
processor number [10] idle for 2 TUS
processor [2] assigned task [23]
processor [3] assigned task [24]
processor [4] assigned task [36]
processor [5] assigned task [39]
processor [6] assigned task [41]

ORIGINAL PAGE IS
OF POOR QUALITY

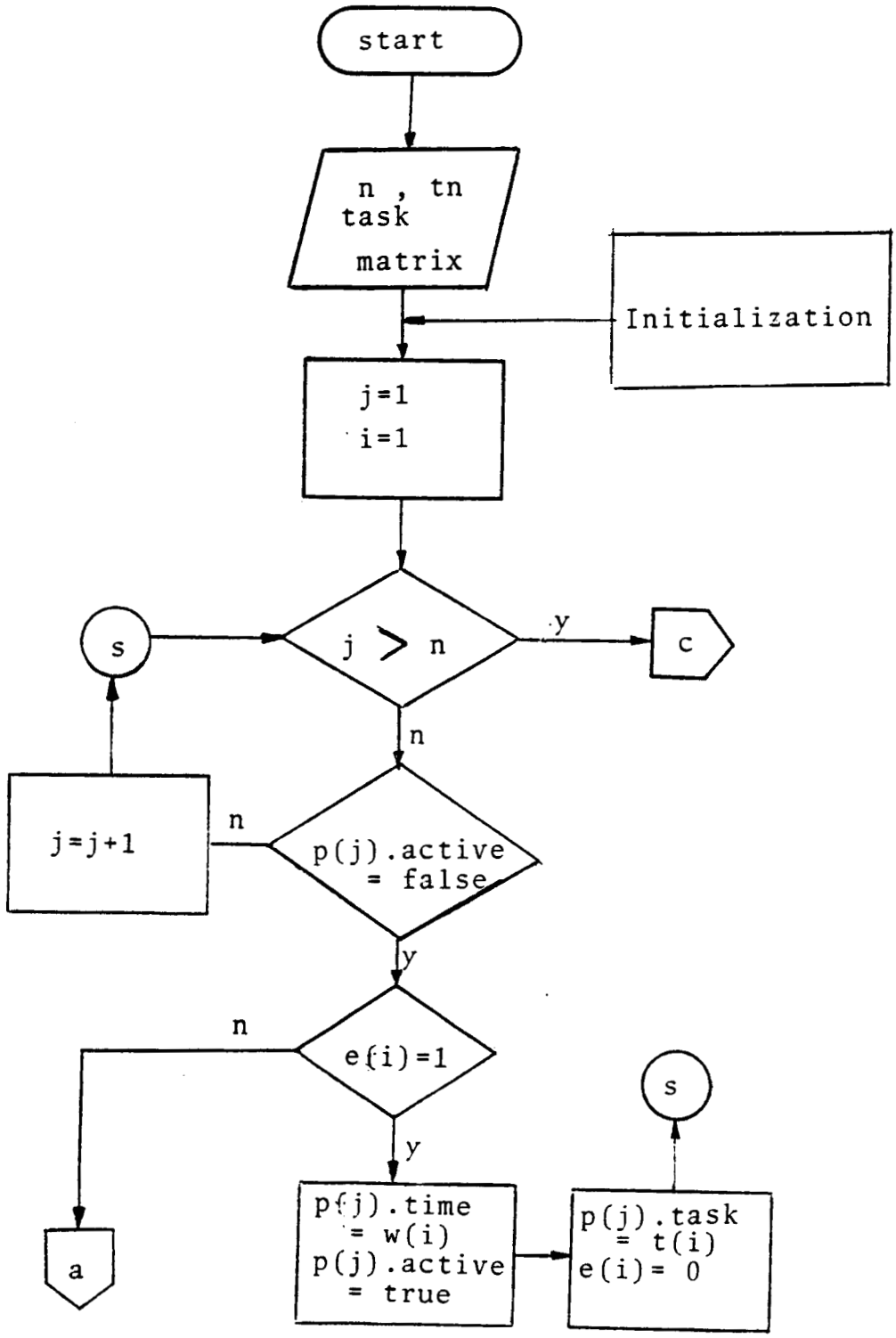
```
processor number [7] idle for 2 TUS
processor number [9] idle for 1 TUS
processor number [10] idle for 3 TUS
processor [7] assigned task [40]
processor number [8] idle for 1 TUS
processor number [9] idle for 2 TUS
processor number [10] idle for 4 TUS
processor [1] assigned task [33]
processor [2] assigned task [43]
processor number [4] idle for 1 TUS
processor number [6] idle for 1 TUS
processor number [8] idle for 2 TUS
processor number [9] idle for 3 TUS
processor number [10] idle for 5 TUS
processor [3] assigned task [34]
processor [4] assigned task [45]
processor number [5] idle for 1 TUS
processor number [6] idle for 2 TUS
processor number [7] idle for 1 TUS
processor number [8] idle for 3 TUS
processor number [9] idle for 4 TUS
processor number [10] idle for 6 TUS
processor number [2] idle for 1 TUS
processor number [5] idle for 2 TUS
processor number [6] idle for 3 TUS
processor number [7] idle for 2 TUS
processor number [8] idle for 4 TUS
processor number [9] idle for 5 TUS
processor number [10] idle for 7 TUS
processor [1] assigned task [42]

Schedule Complete
```



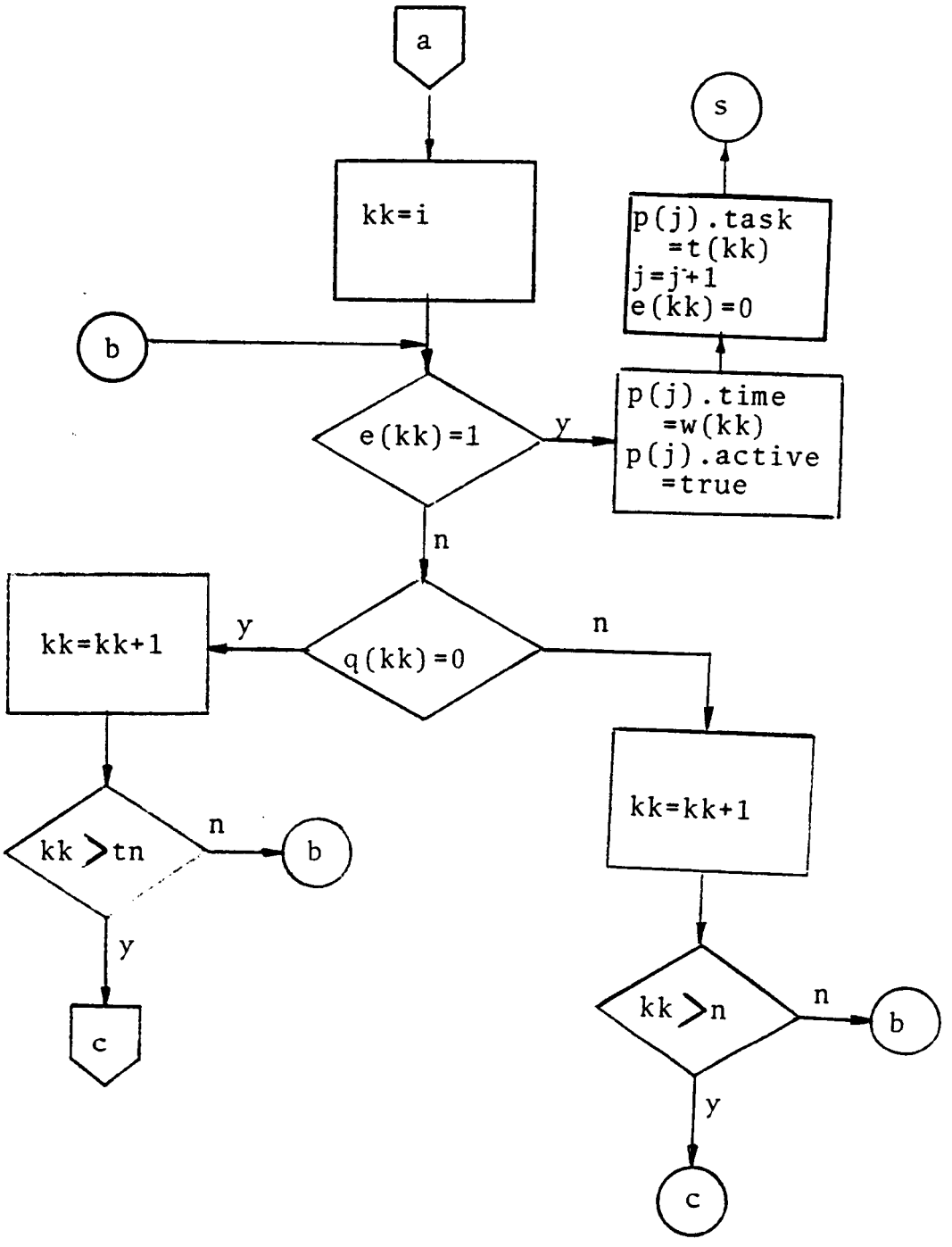
Gantt Chart of Optimal Schedule

APPENDIX C

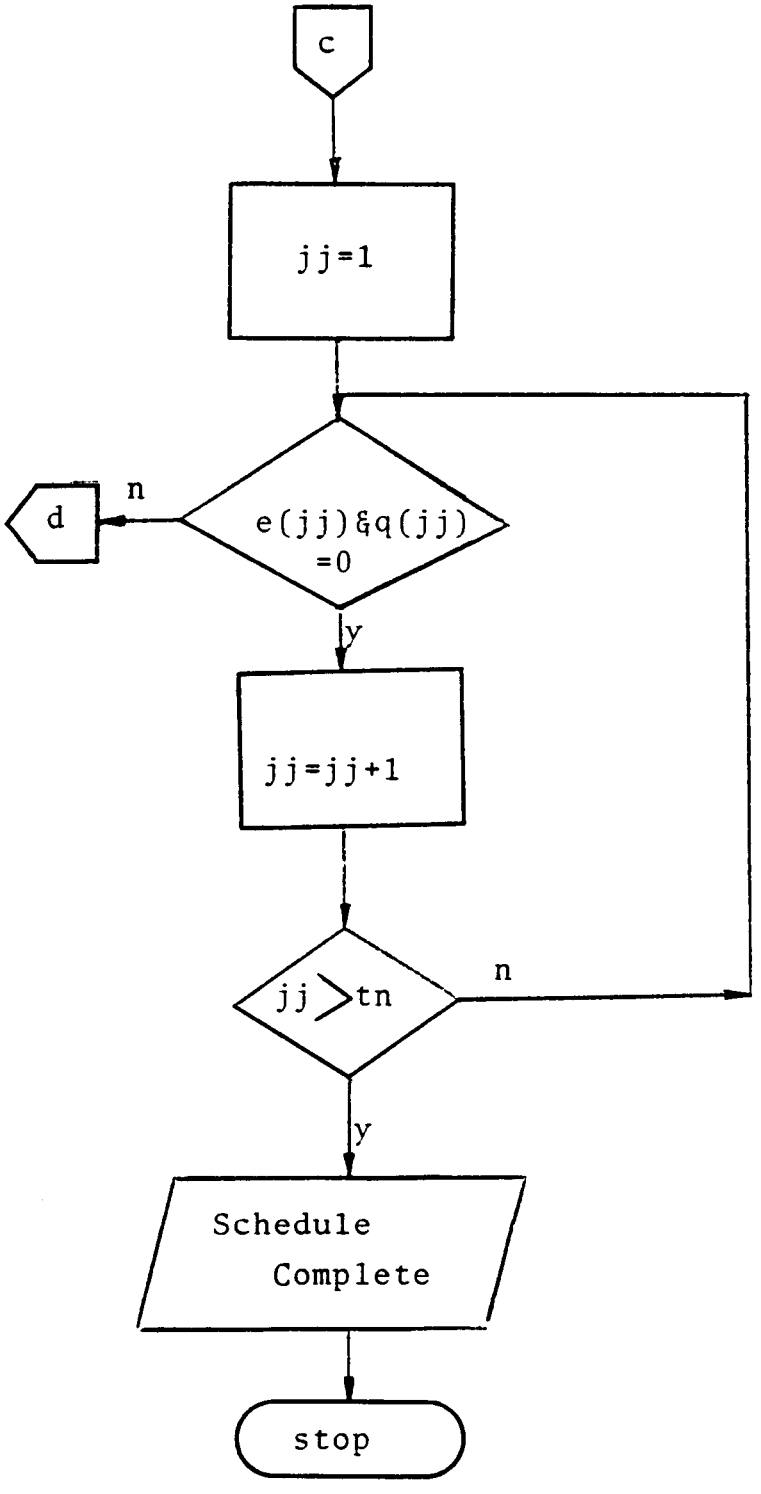
FLOWCHART FOR SCHEDULING ALGORITHM


start
n , tn
task
matrix
Initialization
j=1
i=1
s
j > n
y
c
n
j=j+1
n
p(j).active
= false
y
n
e(i)=1
y
s
a
p(j).time
= w(i)
p(j).active
= true
p(j).task
= t(i)
e(i)= 0

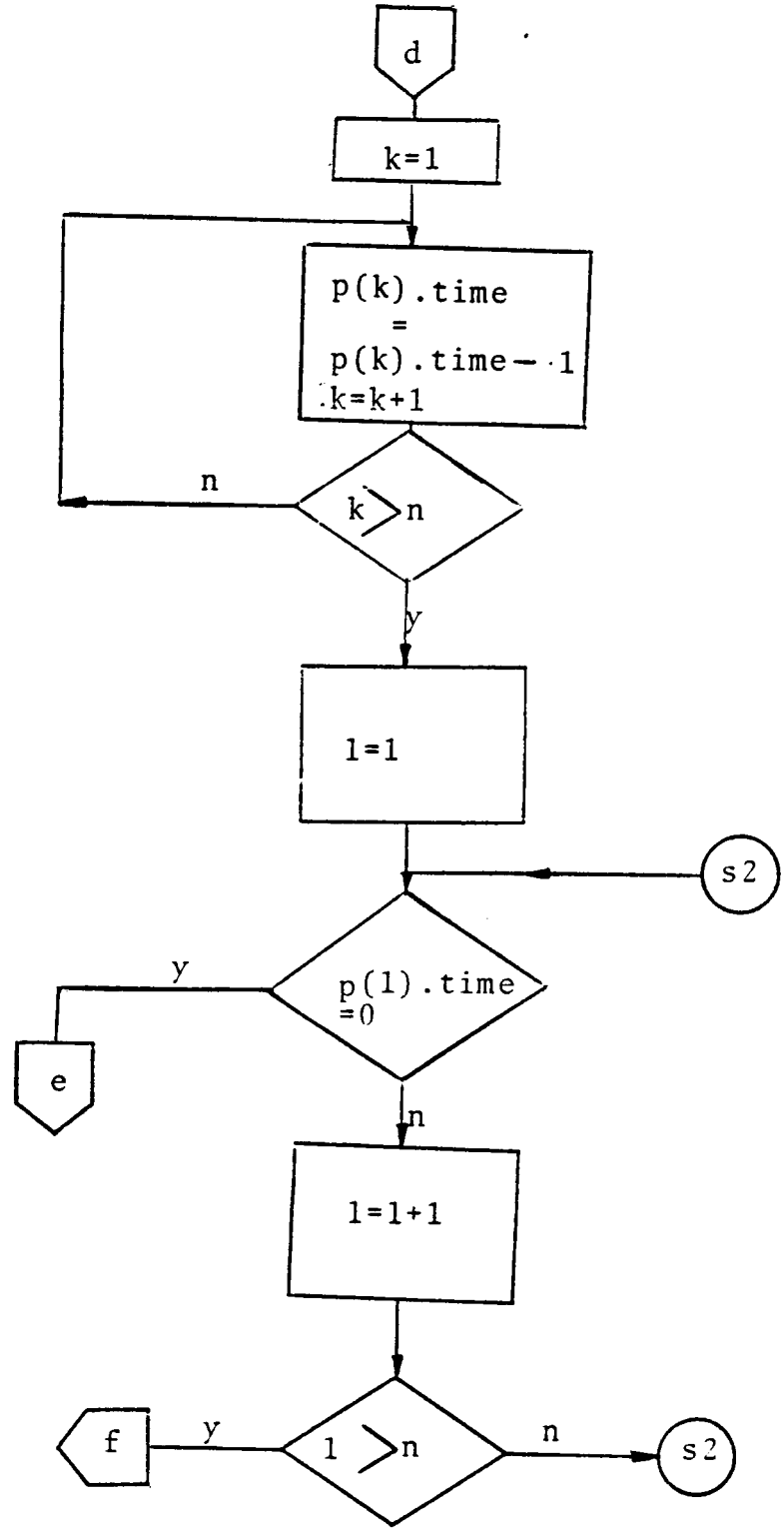
(CONTINUED)



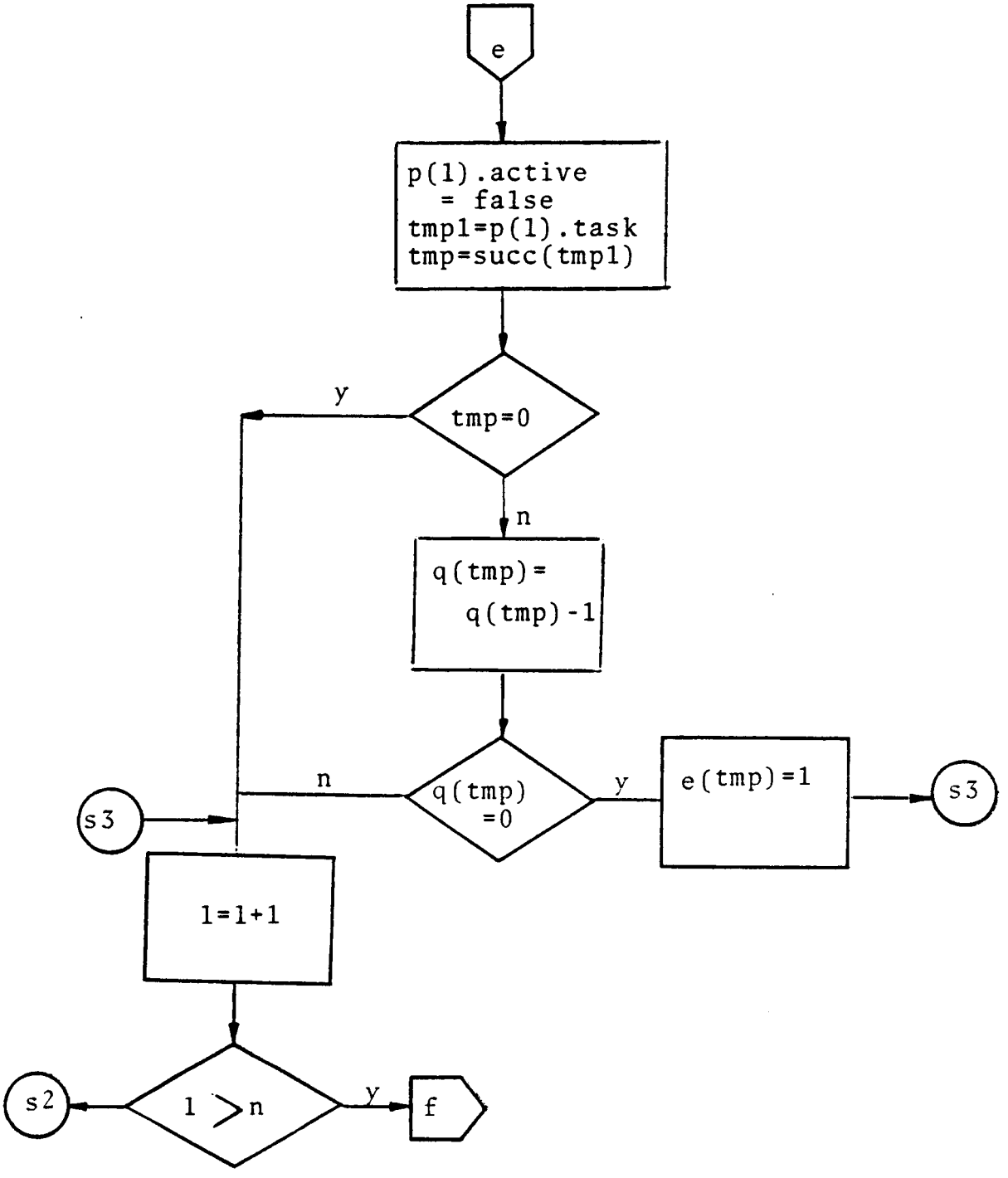
Flow Chart for Procedure 'Schedule'



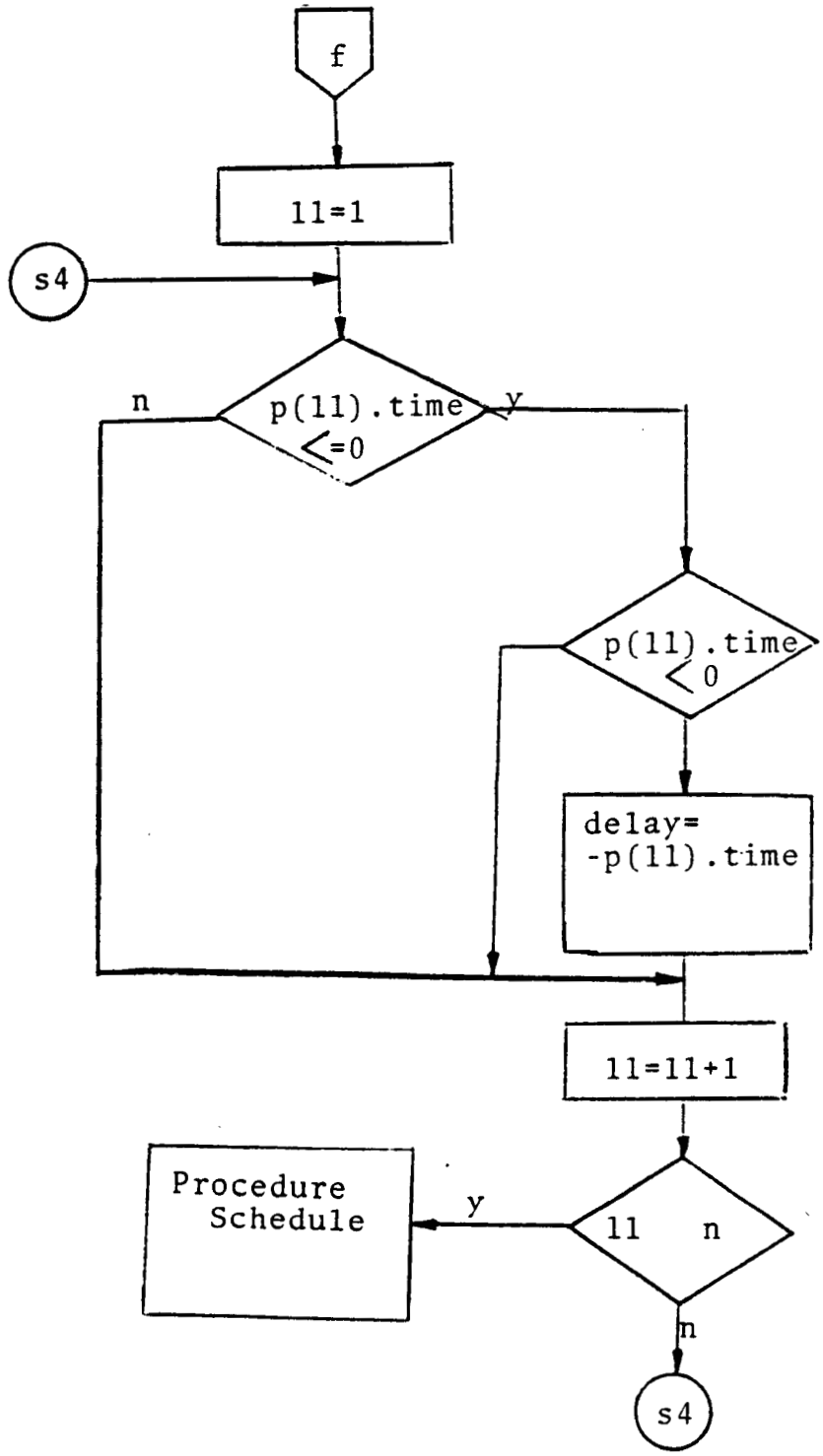
Flow Chart for Procedure 'Check_Schedule'



(CONTINUED)



Flowchart for Procedure 'Time_Processing'



Flowchart for Procedure Reallocate

APPENDIX D

SCHEDULER ROUTINE IN PASCAL

```
{*************************************************************
The following Pascal routine allocates tasks to a set of
processors inaccordance with the scheduling algorithm
already outlined in Chapter 3. The number of processing
elements is treated as a variable parameter.The program
requires as input the following:
        1) The number of available PEs denoted by "n"
        2) The number of defined tasks denoted by "tn"
        3) The task matrix which is read from an input
           data file
The program outputs the delay time of each processor and
also the task numbers which are assigned to a particular
processor. It keeps track of the time schedule of each
processor by providing relevant information.
*************************************************************}


 program processor_scheduling;


const
  max_succ=7;
{ max_succ is the maximum number of successors that can
be present at each node of the task graph. It can be
predefined to assume any value.In this case it has been
defined to be equal to seven as this is adequate for the
task graph under consideration. }

type
processor=record
   time:integer;    { Each processor is defined as a record }
   task:integer;    { the boolean field denotes whether a   }
   active:boolean; { processor is active / inactive         }
end;
proc= array[1..20] of processor;{ maximum number of PEs }
arraytype= array[1..50] of integer;
successorarray=array[1..50,1..50] of integer;


var
  ii,tn,n,inp,z,is:integer;
  e,q,w,t:arraytype;
  suc:successorarray;
  p:proc;
  filvar1,filvar2:text;
  f1,f2:string[12];
```

```
procedure INITIALISE;

{*****************************************************
 This procedure intialises all the PEs by making the
 active field false and setting task time and number = 0.
 It provides the scheduler with a set of PEs that are
 ready to be assigned to incumbent tasks.
 *****************************************************}

 var
   ki:integer;

 begin
    for ki:=1 to n do
      begin
         p[ki].time:=0;
         p[ki].task:=0;
         p[ki].active:=false;
      end;
 end;



 procedure SCHEDULE;


 {*********************************************************
 This procedure allocates a set of available tasks to a
 set of processors that are inactive or available. After
 initial assignment, it checks whether all tasks have been
 scheduled by invoking the procedure check_schedule.
 *********************************************************}


 label
    start,mark;

 var
    i,j,kk:integer;
```

```
procedure TIME_PROCESSING;

{*****************************************************
 This procedure decrements the time field of each
 processor and after each decrement makes a self check
 to ascertain whether any processor is idle. If all
 processors are active then it continues decrementing.
 If any processor is idle, it invokes the procedure
 reallocate for reallocation of any available task to
 the idle processor / processors.
 *****************************************************}

 label
    s1,s2;

 var
    k,l,temp1,temp,jkk,no_succ,max_it:integer;



procedure REALLOCATE;

{*****************************************************
This procedure handles situations when some processors
become free due to task completion while some are still
active. The idle processors are assigned to incumbent
tasks. If no tasks are available, then idle time
is recorded for the inactive processors. After possible
reallocation, the main scheduling program is again invoked.
*****************************************************}

label
    f1;

var
    l1,delay:integer;
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
      begin  { of REALLOCATE }

         l1:=1;
         f1:if p[l1].time <= 0 then
              begin
                 if p[l1].time < 0 then
                   begin
                     delay:= -(p[l1].time);
writeln(filvar2,' processor number [',l1,'] idle for ',delay, ' TUS');
                   end;
                   l1:=l1+1;
                   if l1 > n then
                   SCHEDULE
                   else goto f1 ;
              end
            else
              begin
                l1:=l1+1;
                if l1 > n then
                  begin
                     SCHEDULE;
                  end
                else
                  goto f1;
                  end;
              end;     { of REALLOCATE }



      begin  { of TIME_PROCESSING }
         k:=1;
         s1: p[k].time:=p[k].time-1;
         k:=k+1;
         if k > n then
           begin
              l:=1;
              s2:if p[l].time = 0 then
                 begin
                    p[l].active:=false;
                    temp1:=p[l].task;
                    no_succ:= suc[temp1,1];
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
                max_it:=no_succ+1;
                for jkk:=2 to max_it do
                 begin
                   temp:=suc[temp1,jkk];
                   if temp <> 0 then
                     begin
                       q[temp]:=q[temp]-1;
                        if q[temp]=0 then e[temp]:=1;
                     end;
                 end;
                 l:=l+1;
                 if l > n then
                     REALLOCATE
                 else
                   goto s2;
               end
             else
              begin
               l:=l+1;
               if l > n then
                 begin
                    REALLOCATE;
                 end
               else
                 goto s2;
              end;
           end
        else
          begin
            goto s1;
          end;
end;
```

```
 procedure CHECK_SCHEDULE;


{*********************************************************
This procedure examines the task matrix to ensure that
scheduling is complete, that is, the task graph has been
completely executed. If not, it invokes procedure
time_processing to begin task execution once again.
If allocation is complete, it indicates this by displaying
"Schedule Complete."
 *********************************************************}

 label
     11;

 var
   jj: integer;

 begin
    jj:=1;
    11:if( e[jj]=0) and (q[jj]=0) then
     begin
      jj:=jj+1;
      if jj > tn then
       begin
          writeln (filvar2,'Schedule Complete');
       end
      else
       begin
          goto 11;
       end;
    end
  else
    begin
       TIME_PROCESSING;
    end;
end;
```

ORIGINAL PAGE IS
OF POOR QUALITY

```
begin  { of SCHEDULE }
   i:=1;
   j:=1;
   start: if j > n then
     begin
        CHECK_SCHEDULE;
     end
   else
     begin
        if p[j].active = false then
          begin
            if e[i]=1 then
              begin
                p[j].time:=w[i];
                p[j].active:=true;
                p[j].task:=t[i];
                e[i]:=0;
     writeln(filvar2,' processor [',j,'] assigned task [',i,']');
                i:=i+1;
                j:=j+1;
                goto start;
              end
           else
              begin
                kk:=i;
                mark: if e[kk]=1 then
                  begin
                     p[j].time:=w[kk];
                     p[j].active:=true;
                     p[j].task:=t[kk];
                     e[kk]:=0;
    writeln(filvar2,' processor [',j,'] assigned task [',kk,']');
                     j:=j+1;
                     goto start;
                  end
                else
                   begin
                      if q[kk] =0 then
                        begin
                           kk:=kk+1;
```

```
                        if kk > tn  then
                          begin
                              CHECK_SCHEDULE;
                          end
                        else
                          begin
                             goto mark;
                          end;
                    end
                 else
                    begin
                    kk:=kk+1;
                    if kk > tn then
                      begin
                         CHECK_SCHEDULE;
                      end
                    else
                      begin
                         goto mark;
                      end;
                    end;
               end;
      end;
     end
    else
      begin
         j:=j+1;
         goto start;
      end;
  end;
end;
```

ORIGINAL PAGE IS
OF POOR QUALITY

ORIGINAL PAGE IS
OF POOR QUALITY

```
begin  {of MAIN}
    writeln('input number of processors');
    readln(n);
    writeln(' SELECT INPUT DATA FILE ');
    writeln(' OPTIONS-T1.DAT/T2.DAT ');
    readln(f1);
    assign(filvar1,f1);
    reset(filvar1);
    writeln(' SELECT OUTPUT DATA FILE ');
    writeln(' OPTIONS- R1.DAT/R2.DAT ');
    readln(f2);
    assign(filvar2,f2);
    rewrite(filvar2);
    writeln(filvar2,' TASK ALLOCATION ');
    writeln(filvar2, 'THE NUMBER OF PROCESSORS USED=',n);
    readln(filvar1,tn);
    writeln(filvar2, 'THE NUMBER OF DEFINED TASKS=', tn);
    for ii:=1 to tn do
    begin
       t[ii]:=ii;
    end;
    for ii:=1 to tn do
      begin
         readln(filvar1,e[ii]);
      end;
      for ii:=1 to tn do
        begin
           readln(filvar1,q[ii]);
        end;
        for ii:=1 to tn do
          begin
            for is:=1 to max_succ do
              begin
                readln(filvar1,suc[ii,is]);
              end;
          end;
          for ii:=1 to tn do
            begin
               readln(filvar1,w[ii]);
            end;
            INITIALISE;
            SCHEDULE;
            close(filvar2);
end.
```